conan Documentation

Release 1.3.3

conan

Sep 27, 2019

CONTENTS

Conan is a portable package manager, intended for C and C++ developers, but it is able to manage builds from source, dependencies, and precompiled binaries for any language.

For more information, check conan.io.

Contents:

CHAPTER

ONE

UPGRADING TO CONAN 1.0

If you were using a 0.X conan version, there are some things to consider while upgrading. They are reflected in the *changelog*., but this section summarizes the most important changes here:

1.1 Command line changes

There has been a few things that will break existing usage (compared to 0.30). Most of them are in command line arguments, so they are relatively easy to fix. The most important one is that now most commands require the path to the conanfile folder or file, instead of using `--path` and `--file` arguments. Specifically, **`conan install`**, **`conan export`** and **`conan create`** will be the ones most affected:

```
# instead of --path=myfolder --file=myconanfile.py, now you can do:
$ conan install . # Note the "." is now mandatory
$ conan install folder/myconanfile.txt
$ conan install ../myconanfile.py
$ conan info .
$ conan create . user/channel
$ conan create . Pkg/0.1@user/channel
$ conan create mypkgconanfile.py Pkg/0.1@user/channel
$ conan export . user/channel
$ conan export . Pkg/0.1@user/channel
$ conan export myfolder/myconanfile.py Pkg/0.1@user/channel
```

This behavior aligns with the **`conan source`**, **`conan build`**, **`conan package`** commands, that all use the same arguments to locate the *conanfile.py* containing the logic to be run.

Now all commands read: **`command <origin-conanfile> ...`**

Also, all arguments to command line now use dash instead of underscore:

```
$ conan build .. --source-folder=../src  # not --source_folder
```

1.2 Deprecations/removals

- scopes were completely removed in conan 0.30.X
- `self.conanfile_directory` has been removed. Use `self.source_folder`, `self.build_folder`, etc. instead
- `self.cpp_info`, `self.env_info` and `self.user_info` scope has been reduced to only the `package_info()` method

- `gcc` and `ConfigureEnvironment` were already removed in conan 0.30.1
- `werror` doesn't exist anymore. Now it is the builtin behavior.
- Command `test_package` has been removed. Use **`conan create`** and **`conan test`** instead.
- `CMake` helper only allows now (from conan 0.29). the `CMake(self)` syntax
- **`conan package_files`** command was replaced in conan 0.28 by **`conan export-pkg`** command.

1.3 Settings and profiles. Gcc/CLang versioning

gcc and clang compilers have modified their versioning approach, from gcc > 5 and clang > 4, the minors are really bugfixes, and then they have binary compatibility. To adapt to this, conan now includes major version in the *settings.yml* default settings file:

```
gcc:
    version: ["4.1", "4.4", "4.5", "4.6", "4.7", "4.8", "4.9",
              "5", "5.1", "5.2", "5.3", "5.4",
              "6", "6.1", "6.2", "6.3", "6.4",
              "7", "7.1", "7.2"]
```

Most package creators want to use the major-only settings, like `-s compiler=gcc -s compiler.version=5`, instead of specifying the minors too.

The default profile detection and creation has been modified accordingly, but if you have a default profile you may want to update it to reflect this:

Conan associated tools (conan-package-tools, conan.cmake) have been upgraded to accomodate this new defaults.

1.4 New features

- Cross-compilation support with new default settings in settings.yml: `os_build`, `arch_build`, `os_target`, `arch_target`. They are automatically removed from the `package_id` computation, or kept if they are the only ones defined (as it happens usually with dev-tools packages). It is possible to keep them too with the `self.info.include_build_settings()` method (call it in your `package_id()` method).

Important: Please **don't** use cross-build settings `os_build`, `arch_build` for standard packages and libraries. They are only useful for packages that are used via `build_requires`, like `cmake_installer` or `mingw_installer`.

- Model and utilities for Windows subsystems

```
os:
    Windows:
        subsystem: [None, cygwin, msys, msys2, wsl]
```

This subsetting can be used by build helpers as `CMake`, to act accordingly.

CHAPTER

TWO

INTRODUCTION

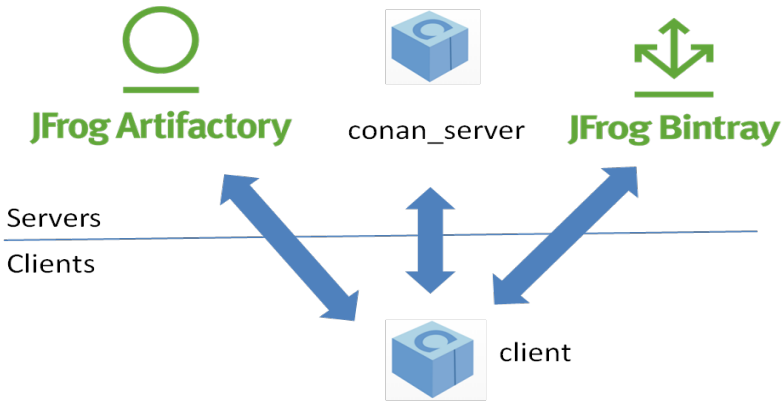
2.1 Open Source

Conan is OSS, with an MIT license. Check out the source code and issue tracking (for reporting bugs and for feature requests) at https://github.com/conan-io/conan

2.2 Decentralized package manager

Conan is a decentralized package manager with a client-server architecture. This means that clients can fetch packages from, as well as upload packages to, different servers ("remotes"), similar to the "git" push-pull model to/from git remotes.

On a high level, the servers are just package storage. They do not build nor create the packages. The packages are created by the client, and if binaries are built from sources, that compilation is also done by the client application.



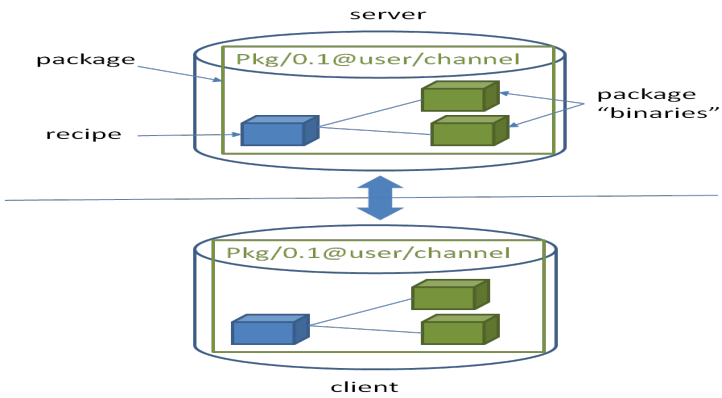
The different applications in the image above are:

- The **conan client**: this is a console/terminal command line application, containing the heavy logic for package creation and consumption. Conan client has a local cache for package storage, and so it allows you to fully create and test packages offline. You can also **work offline** so long as no new packages are needed from remote servers.
- The **conan server**: this is a TCP server that can be easily run as your **own server on-premises** to host your private packages. It is also a service application that can be run as a daemon or service, behind a web server (apache, nginx), or easily as stand-alone application. Both the conan client and the conan_server are OSS, MIT license, so you can use them for free in your company, customize them, or redistribute them without any legal issue.

- **JFrog** Artifactory offers conan repositories; so it can also be used as an on-premises server. It is a more powerful solution, featuring a WebUI, multiple auth protocols, High Availability, etc. It also has cloud offerings that will allow you to have private packages without having any on-premises infrastructure.
- **JFrog** Bintray provides a public and free hosting service for OSS conan packages. Users can create their own repositories under their accounts and organizations, and freely upload conan packages there, without moderation. You should, however, take into account that those packages will be public, and so they must conform to the respective licenses, especially if the packages contain third party code. Just reading or retrieving conan packages from Bintray, doesn't require an account, an account is only needed to upload packages. Besides that, Bintray provides a central repository called conan-center which is moderated, and packages are reviewed before being accepted to ensure quality.

2.3 Binary management

One of the most powerful features of Conan is that it can manage pre-compiled binaries for packages. To define a package, referenced by its name, version, user and channel, a package recipe is needed. Such a package recipe is a conanfile.py python script that defines how the package is built from sources, what the final binary artifacts are, the package dependencies, etc.



When a package recipe is used in the Conan client, and a "binary package" is built from sources, that binary package will be compatible with specific settings, such as the OS it was created for, the compiler and compiler version, or the computer architecture. If the package is built again from the same sources but with different settings, (e.g. for a different architecture), a new, different binary will be generated. By the way, "binary package" is in quotes because, strictly, it is not necessarily a binary. A header-only library, for example, will contain just the headers in the "binary package".

All the binary packages generated from a package recipe are managed and stored coherently. When they are uploaded to a remote, they stay connected. Also, different clients building binaries from the same package recipe (like CI build slaves in different operating systems), will upload their binaries under the same package name to the remotes.

Package consumers (client application users that are installing existing packages to reuse in their projects) will typically retrieve pre-compiled binaries for their systems in case such compatible binaries exist. Otherwise those packages will be built from sources on the client machine to create a binary package matching their settings.

2.4 Cross platform, build system agnostic

Conan works and is being actively used on Windows, Linux (Ubuntu, Debian, RedHat, ArchLinux, Raspbian), OSX, FreeBSD, and SunOS, and, as it is portable, it might work in any other platform that can run python. In the documentation, examples for a specific OS might be found, such as `conan install . -s compiler="Visual`

Studio", which will be specific for Windows users. If on a different system, the reader should adapt to their own platform and settings (for example **conan install . -s compiler=gcc**).

Also **conan works with any build system**. In the documentation, CMake will be widely used, because it is portable and well known. But conan does not depend on CMake at all; it is not a requirement. **Conan is totally orthogonal to the build system**. There are some utilities that improve the usage of popular build systems such as CMake or Autotools, but they are just helpers. Furthermore, it is not necessary that all the packages are built with the same build system. It is possible to depend on packages created with other build system than the one you are using to build your project.

2.5 Stable

From conan 1.0, there is a commitment to stability, not breaking user space while evolving the tool and the platform. This means:

- Moving forward to following minor versions 1.1, 1.2, . . . , 1.X should never break existing recipes, packages or command line flows
- If something is breaking, it will be considered a bug and reverted
- Bug fixes will not be considered breaking, recipes and packages relying on the incorrect behavior of such bug will be considered already broken.
- Only documented features are considered part of the public interface of conan. Private implementation details, and everything not included in the documentation is subject to change.
- Configuration and automatic tools detection, like the detection of the default profile might be subject to change. Users are encouraged to define their configurations in profiles for repeatability. New installations of conan might use different configuration.

The compatibility is always considered forward. New APIs, tools, methods, helpers can be added in following 1.X versions. Recipes and packages created with these features will be backwards incompatible with earlier conan versions.

This means that public repositories, like conan-center assume the use of the latest version of the conan client, and using an older version may result in failure of packages and recipes created with a newer version of the client.

If you have any question regarding conan updates, stability, or any clarification about this definition of stability, please report in the documentation issue tracker: https://github.com/conan-io/docs.

Got any doubts? Please check out our *FAQ section* or .

CHAPTER

THREE

INSTALL

Conan can be installed in many Operating Systems. It has been extensively used and tested in Windows, Linux (different distros), OSX, and is also actively used in FreeBSD and Solaris SunOS. There are also several additional operating systems on which it has been reported to work.

There are three ways to install conan:

1. The preferred and **strongly recommended way to install Conan** is from PyPI, the Python Package Index, using the `pip` command.
2. There are other available installers for different systems, which might come with a bundled python interpreter, so that you don't have to install python first. Please note that some of **these installers might have some limitations**, specially those created with pyinstaller (such as Windows exe & Linux deb).
3. Running conan from sources.

3.1 Install with pip (recommended)

To install Conan using `pip`, you need a python 2.7 or 3.X distribution installed in your machine. Modern python distros come with pip pre-installed. However, if necessary you can install pip by following the instructions in pip docs.

Warning: Python 2 will by deprecated soon by the Python maintainers. It is strongly recommended to use Python 3 for conan, especially if need to manage non-ascii filenames or file contents. Conan still supports Python 2, but some of the dependencies have started to be Python 3 only too. The roadmap for deprecating Python 2 support in Conan will be defined soon.

Install conan:

```
$ pip install conan
```

Important: Please READ carefully

- Make sure that your **pip** installation matches your **python (2.7 or 3.X)** one.
- In Linux if you want to install it globally, you might need **sudo** permissions.
- We strongly recommend using **virtualenvs** (virtualenvwrapper works great) for everything python related
- In **Windows** and with Python 2.7, you might need to use **32bits** python distribution (which is the Windows default one), instead of 64 bits.

- In **OSX**, specially latest versions that might have **System Integrity Protection**, pip might fail. Try with virtualenvs, or install with other user `$ pip install --user conan`.
- If you are in Windows, and using python <3.5, you might have problems if python is installed in a path with spaces, like "C:/Program Files(x86)/Python". This is a known python's limitation, not Conan's. Install python in a path without spaces, use a virtualenv in another location or upgrade your python installation.
- In some Linux distros, like Linux Mint, it is possible that you need a restart (shell restart, or logout/system if not enough) after installation, so Conan is found in the path.
- Windows, Python 3 installation can fail installing the `wrapt` dependency because a bug in **pip**. Information about the issue and workarounds is here: https://github.com/GrahamDumpleton/wrapt/issues/112.

3.2 Install from brew (OSX)

There is a brew recipe, so in OSX, you can install Conan as follows:

```
$ brew update
$ brew install conan
```

3.3 Install from AUR (Arch Linux)

You can find the package here. The easiest way is using **pacaur** tool:

```
$ pacaur -S conan
```

Or you can also use `makepkg` and install it following the AUR docs: installing packages.

Just remember to install four Conan dependencies first. They are not in the official repositories but there are in **AUR** repository too:

- python-patch
- python-node-semver
- python-distro
- python-pluginbase

3.4 Install the binaries

Go to the conan website and download the installer for your platform!

Execute the installer. You don't need to install python.

3.5 Initial configuration

Let's check if conan is correctly installed. In your console, run the following:

```
$ conan
```

You will see something similar to:

```
Consumer commands
  install    Installs the requirements specified in a conanfile (.py or .txt).
  config     Manages configuration. Edits the conan.conf or installs config files.
  get        Gets a file or list a directory of a given reference or package.
  info       Gets information about the dependency graph of a recipe.
  ...
```

3.6 Install from source

You can run conan directly from source code. First you need to install Python 2.7 or Python 3 and pip.

Clone (or download and unzip) the git repository and install its requirements:

```
$ git clone https://github.com/conan-io/conan.git
$ cd conan
$ pip install -r conans/requirements.txt
```

Create a script to run Conan and add it to your `PATH`.

```
#!/usr/bin/env python

import sys

conan_repo_path = "/home/your_user/conan" # ABSOLUTE PATH TO CONAN REPOSITORY FOLDER

sys.path.append(conan_repo_path)
from conans.client.command import main
main(sys.argv[1:])
```

Test your `conan` script.

```
$ conan
```

You should see the Conan commands help.

CHAPTER

FOUR

GETTING STARTED

Let's start with an example using one of the most popular C++ libraries: POCO. For convenience purposes we'll use CMake. Keep in mind that Conan **works with any build system** and does not depend on CMake.

4.1 A Timer using POCO libraries

First, let's create a folder for our project:

```
$ mkdir mytimer
$ cd mytimer
```

Note: If your code is in a GitHub repository you can simply clone the project, instead of creating this folder, using the following command:

```
$ git clone https://github.com/memsharded/example-poco-timer.git mytimer
```

Next, create the following source files inside this folder:

Listing 1: **timer.cpp**

```
// $Id: //poco/1.4/Foundation/samples/Timer/src/Timer.cpp#1 $
// This sample demonstrates the Timer and Stopwatch classes.
// Copyright (c) 2004-2006, Applied Informatics Software Engineering GmbH.
// and Contributors.
// SPDX-License-Identifier:    BSL-1.0

#include "Poco/Timer.h"
#include "Poco/Thread.h"
#include "Poco/Stopwatch.h"
#include <iostream>

using Poco::Timer;
using Poco::TimerCallback;
using Poco::Thread;
using Poco::Stopwatch;

class TimerExample{
public:
    TimerExample(){ _sw.start();}
```

(continues on next page)

(continued from previous page)

```
    void onTimer(Timer& timer){
        std::cout << "Callback called after " << _sw.elapsed()/1000 << "␣
↪milliseconds." << std::endl;
    }
private:
    Stopwatch _sw;
};

int main(int argc, char** argv){
    TimerExample example;
    Timer timer(250, 500);
    timer.start(TimerCallback<TimerExample>(example, &TimerExample::onTimer));

    Thread::sleep(5000);
    timer.stop();
    return 0;
}
```

Now create a *conanfile.txt* inside this folder with the following content:

Listing 2: **conanfile.txt**

```
[requires]
Poco/1.9.0@pocoproject/stable

[generators]
cmake
```

In this example we will use CMake to build the project, which is why the `cmake` generator is specified. This generator will create a *conanbuildinfo.cmake* file that defines CMake variables as include paths and library names, that can be used in our build.

Note: If you are not a CMake user, change the `[generators]` section of your *conanfile.txt* to `gcc` or a more generic one `txt` to handle requirements with any build system. Learn more in *Using packages*.

Just include the generated file and use these variables inside our *CMakeLists.txt*:

Listing 3: **CMakeLists.txt**

```
project(FoundationTimer)
cmake_minimum_required(VERSION 2.8.12)
add_definitions("-std=c++11")

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

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

4.2 Installing dependencies

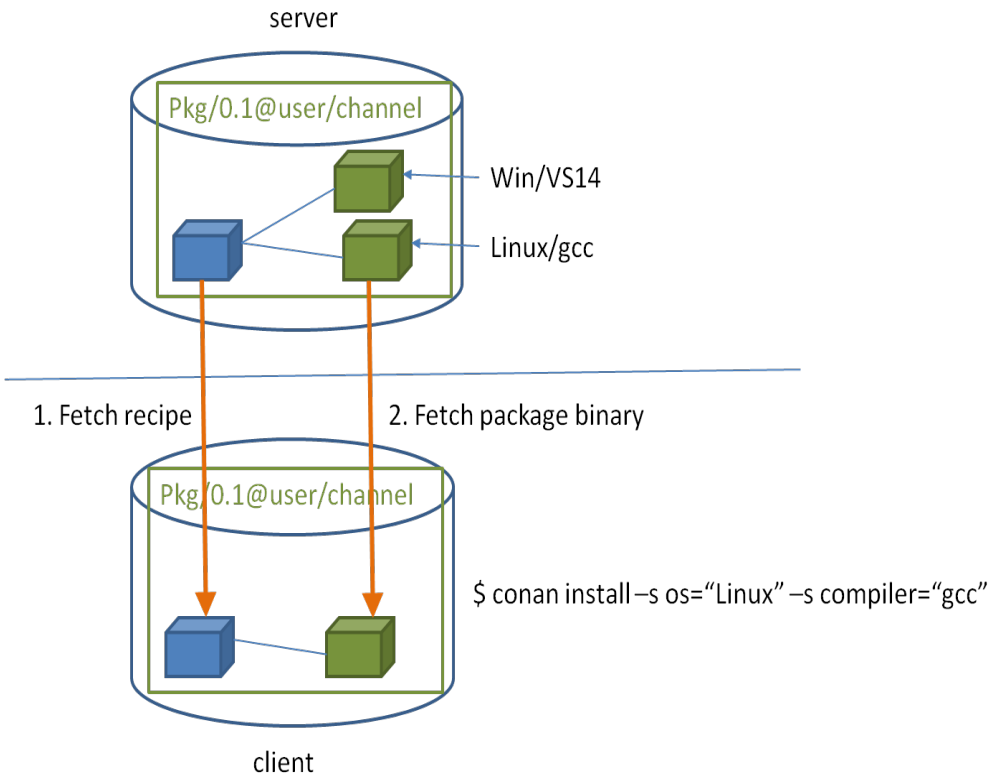
If you have a terminal with light colors, like the default gnome terminal in Ubuntu, set `CONAN_COLOR_DARK=1` to have a better contrast. Then create a build folder, for temporary build files, and install the requirements (pointing to

the parent directory, as it is where the *conanfile.txt* is):

```
$ mkdir build && cd build
$ conan install ..
```

This **conan install** command will download the binary package required for your configuration (detected the first time that you ran the command), **together with other (transitively required by Poco) libraries, like OpenSSL and Zlib**. It will also create the *conanbuildinfo.cmake* file in the current directory, in which you can see the cmake defined variables, and a *conaninfo.txt* where information about settings, requirements and options is saved.

It is very important to understand the installation process. When **conan install** command is issued, it will use some settings, specified on the command line or taken from the defaults in `<userhome>/.conan/profiles/default` file.



For a command like **conan install . -s os="Linux" -s compiler="gcc"**, the steps are:

- Check if the package recipe (for `Poco/1.9.0@pocoproject/stable` package) exists in the local cache. If we are just starting, the cache will be empty.
- Look for the package recipe in the defined remotes. Conan comes with conan-center Bintray remote by default (you can change that).
- If the recipe exists, Conan client will fetch and store it in your local cache.
- With the package recipe and the input settings (Linux, gcc), Conan client will check in the local cache if the corresponding binary is there, if we are installing for the first time, it won't.
- Conan client will search for the corresponding binary package in the remote, if it exists, it will be fetched.
- Conan client will then finish generating the requested files specified in `generators`.

If the binary package necessary for some given settings doesn't exist, Conan client will throw an error. It is possible to try to build the binary package from sources with the `--build=missing` command line argument to install. A

detailed description of how a binary package is built from sources will be given in a later section.

Warning: In the Bintray repositories there are binaries for several mainstream compilers and versions, such as Visual Studio 12, 14, linux-gcc 4.9 and apple-clang 3.5. If you are using another setup, the command might fail because of the missing package. You could try to change your settings or build the package from source, using the `--build=missing` option, instead of retrieving the binaries. Such a build might not have been tested and eventually fail.

4.3 Building the timer example

Now you are ready to build and run your project:

```
(win)
$ cmake .. -G "Visual Studio 14 Win64"
$ cmake --build . --config Release

(linux, mac)
$ cmake .. -G "Unix Makefiles" -DCMAKE_BUILD_TYPE=Release
$ cmake --build .
...
[100%] Built target timer
$ ./bin/timer
Callback called after 250 milliseconds.
...
```

4.4 Inspecting dependencies

The retrieved packages are installed to your local user cache (typically `.conan/data`), and can be reused from there in other projects. This allows to clean your current project and keep working even without network connection. Search packages in the local cache using:

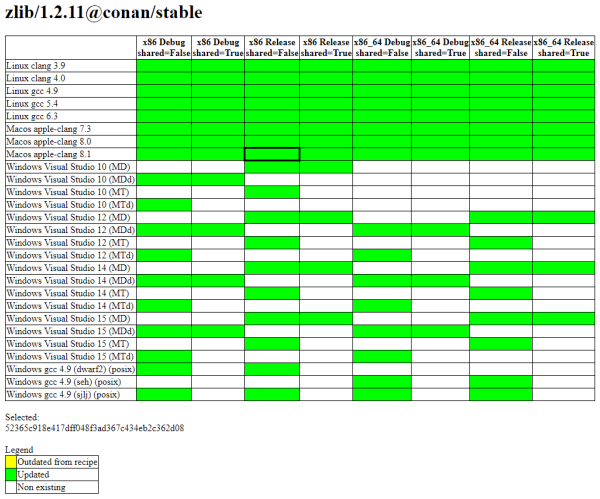
```
$ conan search
```

Inspect binary package details (for different installed binaries for a given package recipe) using:

```
$ conan search Poco/1.9.0@pocoproject/stable
```

There is also the option to generate a table for all binaries from a given recipe with the `--table` option, even in remotes:

```
$ conan search zlib/1.2.11@conan/stable --table=file.html -r=conan-center
$ file.html # or open the file, double-click
```



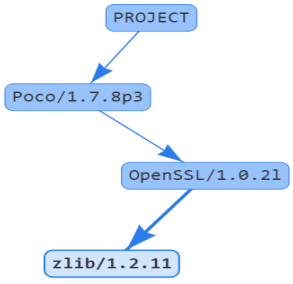
Check the reference for more information on how to search in remotes, how to remove or clean packages from the local cache, and how to define custom cache directory per user or per project.

Inspect your current project's dependencies with the `info` command, pointing it to the folder where the *conanfile.txt* is:

```
$ conan info ..
```

Generate a graph of your dependencies in dot or html formats:

```
$ conan info .. --graph=file.html
$ file.html # or open the file, double-click
```



4.5 Searching packages

The packages that have been used are installed from the remote repository that is configured by default in the conan client, which is called "conan-center" and is in Bintray. You can search for existing packages there with:

```
$ conan search "zlib*" -r=conan-center
```

There are other community repositories that can be configured and used, check them in *this section about remotes*.

4.6 Building with other configurations

In this example we have built our project using the default configuration detected by conan, this configuration is known as the *default profile*.

The first time you run the command that requires a profile, such as **`conan install`**, your settings are detected (compiler, architecture...) automatically and stored as default in a profile. You can change your those settings by editing `~/.conan/profiles/default` or create new profiles with the desired configuration.

Attention:

- It is strongly recommended to review the generated default profile and adjust the settings to describe accurately your system.
- When a GCC **compiler >= 5.1** is detected, the setting modeling the c++ standard library: `compiler.libcxx` will be set to `libstdc++` that represent the old ABI compatibility for better compatibility. Your compiler default is likely the new ABI so you might want to change it to `libstdc++11` to use the new ABI compliant with CXX11 directives. *Read more here*.

For example, if we have a profile with a gcc configutarion for 32 bits in a profile called *gcc_x86*, we could issue the `install` command like this:

```
$ conan install . -pr gcc_x86
```

Tip: Using profiles is strongly recommended. Learn more about them *here*.

However, the user can always override the default profile settings in `install` command with the `-s` parameter. As an exercise, try building your timer project with a different configuration. For example, you could try building the 32 bits version:

```
$ conan install . -s arch=x86
```

This will install a different package, using the `-s arch=x86` setting, instead of the default used previously, that in most cases will be `x86_64`.

To use the 32 bits binaries you will also have to change your project build:

- In Windows, change the CMake invocation accordingly to `Visual Studio 14`.
- In Linux, you have to add the `-m32` flag to your `CMakeLists.txt` with `SET(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -m32")`, and the same to `CMAKE_C_FLAGS`, `CMAKE_SHARED_LINK_FLAGS` and `CMAKE_EXE_LINKER_FLAGS`. This can also be done more easily, automatically with Conan, as we'll see later. - In Mac, you need to add the definition `-DCMAKE_OSX_ARCHITECTURES=i386`.

Got any doubts? Check out our *FAQ section* or .

CHAPTER

FIVE

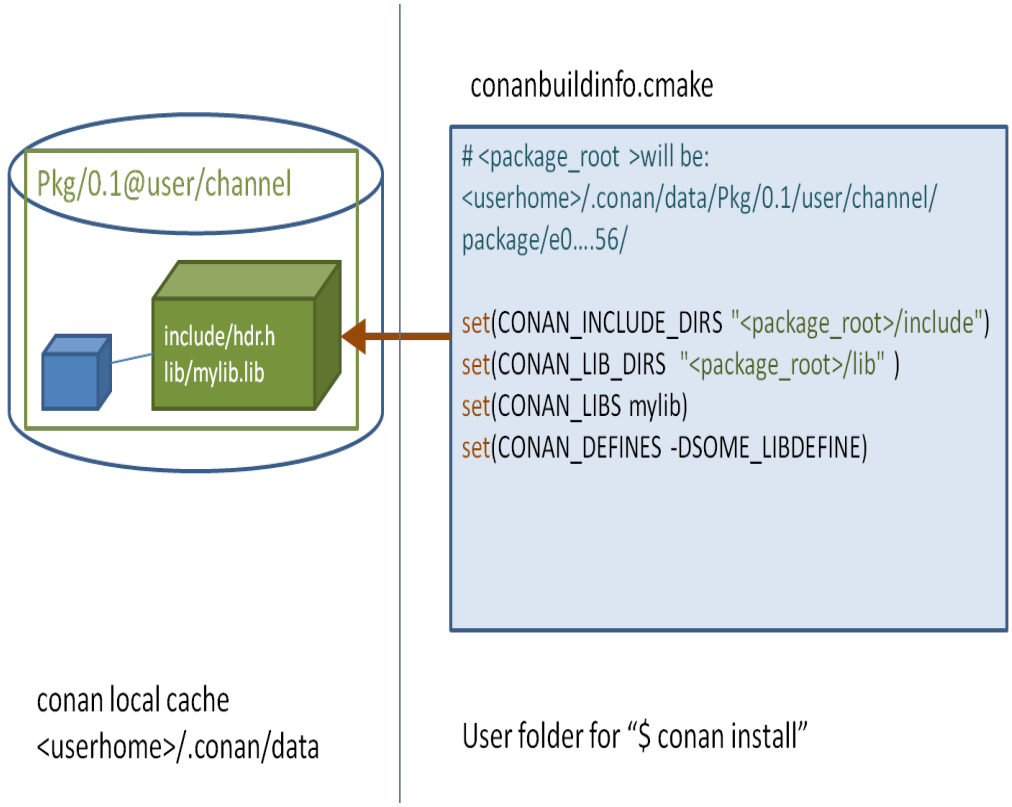
USING PACKAGES

This section shows how to setup your project and manage your dependencies (install existing packages) with conan.

5.1 Installing dependencies

In *Getting started* we used **conan install** command to download the **Poco** library and build an example.

Please take a moment to inspect the generated `conanbuildinfo.cmake` file that was created when we did **conan install**. You can see there that there are many CMake variables declared. For example `CONAN_INCLUDE_DIRS_ZLIB`, which defines the include path to the ZLib headers, or `CONAN_INCLUDE_DIRS` that defines include paths for all dependencies headers.



If you check the full path, you will see that they are pointing to a folder in your `<userhome>` folder, this is called the **local cache**. It is the place where package recipes and binary packages are stored and cached, so they don't have to

be retrieved again. You can inspect the **local cache** with **`conan search`**, and you can also remove packages from it with **`conan remove`** command.

If you navigate to the paths pointed by the `conanbuildinfo.cmake` you will be able to see the headers and the libraries for each package.

If you execute a **`conan install Poco/1.9.0@pocoproject/stable`** command in your shell, conan will download the Poco package and its dependencies (*OpenSSL/1.0.2l@conan/stable* and *zlib/1.2.11@conan/stable*) to your local cache and print information about the folder of the where they are installed. You could handle them manually if you want. But the recommended approach is using a `conanfile.txt`.

5.1.1 Requires

We put the required dependencies in the **[requires]** section. The requirements look like this:

```
[requires]
Poco/1.9.0@pocoproject/stable
```

Where:

- `Poco` is the name of the package, usually the same of the project/library.
- `1.9.0` is the version, usually matching the one of the packaged project/library. Can be any string, not necessarily a number, so it is possible to have a "develop" or "master" version. Packages can be overwritten, so it is also OK to have packages like "nightly" or "weekly", that are regenerated periodically.
- `pocoproject` is the owner of this package version. It is basically a namespace that allows different users to have their own packages for the same library with the same name, and interchange them. So, for example, you can easily upload a certain library under your own user name "lasote", and later those packages can be uploaded without modifications to another official group or company username.
- `stable` is the channel. Channels also allow to have different packages for the same library and use them interchangeably. They usually denote the maturity of the package, as an arbitrary string: "stable", "testing", but it can be used for any purpose, like package revisions (the library version has not changed, but the package recipe has evolved).

Overriding requirements

You can specify multiple requirements and you can **override** the transitive "require's requirements". In our example, conan installed the Poco package and all its requirements transitively:

- **OpenSSL/1.0.2l@conan/stable**
- **zlib/1.2.11@conan/stable**

Tip: This is a good example to explain requirements overriding. We all know the importance of keeping the OpenSSL library updated.

Now imagine that a new release of OpenSSL library is out, and a new conan package for it is available. Do we need to wait until the author pocoproject generates a new package of POCO that includes the new OpenSSL library?

Not necessarily, just enter the new version in **[requires]**:

```
[requires]
Poco/1.9.0@pocoproject/stable
OpenSSL/1.0.2p@conan/stable
```

The second line will override the OpenSSL/1.0.2l required by POCO, with the (non-existent yet) **OpenSSL/1.0.2p**.

Other example could be, in order to try out some new zlib alpha features, we could replace the zlib requirement with one from another user or channel.

```
[requires]
Poco/1.9.0@pocoproject/stable
OpenSSL/1.0.2p@conan/stable
zlib/1.2.11@otheruser/alpha
```

5.1.2 Generators

Conan reads the **[generators]** section from `conanfile.txt` and creates files for each generator with all the necessary information to link your program with the specified requirements. The generated files are usually temporary, created in build folders and not committed to version control, as they have paths to local folder that will not exist in another machine. Also, it is very important to highlight that generated files match the given configuration (Debug/Release, x86/x86_64, etc), specified at **`conan install`** time. If the configuration changes, the files will change.

Check the complete *generators* reference.

5.1.3 Options

We have already seen that there are some **settings** that can be specified at install time, for example **`conan install . -s build_type=Debug`**. The settings are typically a project-wide configuration, defined by the client machine. So they cannot have a default value in the recipe. For example, it doesn't make sense for a package recipe to declare as default compiler "Visual Studio", because that is something defined by the end consumer, and unlikely to make sense if they are working in Linux.

On the other hand, **options** are intended for package specific configuration, that can be set to a default value in the recipe. For example, one package can define that its default linkage is static, and such default will be used if consumers don't specify otherwise.

Note: You can see the available options for a package inspecting the recipe with **`conan get <reference>`** command:

```
$ conan get Poco/1.9.0@pocoproject/stable
```

As an example, we can modify the previous example to use dynamic linkage instead of the default one, which was static. Just edit the *conanfile.txt*:

```
[requires]
Poco/1.9.0@pocoproject/stable

[generators]
cmake

[options]
Poco:shared=True # PACKAGE:OPTION=VALUE
OpenSSL:shared=True
```

Install the requirements and compile from the build folder (change the CMake generator if not in Windows):

```
$ conan install ..
$ cmake .. -G "Visual Studio 14 Win64"
$ cmake --build . --config Release
```

You can also avoid defining the options in the `conanfile.txt` and directly define them in the command line:

```
$ conan install .. -o Poco:shared=True -o OpenSSL:shared=True
# or even with wildcards, to apply to many packages
$ conan install .. -o *:shared=True
```

Conan will install the shared library packages binaries, and the example will link with them. You can again inspect the different installed binaries, e.g. **`conan search zlib/1.2.8@lasote/stable`**.

Finally, launch the executable:

```
$ ./bin/timer
```

What happened? It fails because it can't find the shared libraries in the path. Remember that shared libraries are used at runtime, and should be locatable by the OS, which is the one running the application.

We could inspect the generated executable, and see that it is using the shared libraries. For example in Linux, we could use the *objdump* tool and see in *Dynamic section*:

```
$ cd bin
$ objdump -p timer
...
Dynamic Section:
 NEEDED               libPocoUtil.so.31
 NEEDED               libPocoXML.so.31
 NEEDED               libPocoJSON.so.31
 NEEDED               libPocoMongoDB.so.31
 NEEDED               libPocoNet.so.31
 NEEDED               libPocoCrypto.so.31
 NEEDED               libPocoData.so.31
 NEEDED               libPocoDataSQLite.so.31
 NEEDED               libPocoZip.so.31
 NEEDED               libPocoFoundation.so.31
 NEEDED               libpthread.so.0
 NEEDED               libdl.so.2
 NEEDED               librt.so.1
 NEEDED               libssl.so.1.0.0
 NEEDED               libcrypto.so.1.0.0
 NEEDED               libstdc++.so.6
 NEEDED               libm.so.6
 NEEDED               libgcc_s.so.1
 NEEDED               libc.so.6
```

5.1.4 Imports

There are some differences between shared libraries on linux (*.so), windows (*.dll) and MacOS (*.dylib). The shared libraries must be located in some folder where they can be found, either by the linker, or by the OS runtime.

It is possible to add the folders of the libraries to the path (dynamic linker LD_LIBRARY_PATH path in Linux, DYLD_LIBRARY_PATH in OSX, or system PATH in Windows), or copy those shared libraries to some system folder, so they are found by the OS. But those are typical operations of deploys or final installation of apps, not desired while developing, and conan is intended for developers, so it tries not to mess with the OS.

In Windows and OSX, the simplest approach is just to copy the shared libraries to the executable folder, so they are found by the executable, without having to modify the path.

We can easily do that with the **[imports]** section in `conanfile.txt`. Let's try it.

Edit the `conanfile.txt` file and paste the following **[imports]** section:

```
[requires]
Poco/1.9.0@pocoproject/stable

[generators]
cmake

[options]
Poco:shared=True
OpenSSL:shared=True

[imports]
bin, *.dll -> ./bin # Copies all dll files from packages bin folder to my "bin" folder
lib, *.dylib* -> ./bin # Copies all dylib files from packages lib folder to my "bin"␣
↪folder
```

Note: You can explore the package folder in your local cache (~/.conan/data) and look where the shared libraries are. It is common that ***.dll** are copied in **/bin** the rest of the libraries should be found in the **/lib** folder. But it's just a convention, different layouts are possible.

Install the requirements (from the `mytimer/build` folder), and run the binary again:

```
$ conan install ..
$ ./bin/timer
```

Now look at the `mytimer/build/bin` folder and verify that the needed shared libraries are there.

As you can see, the **[imports]** section is a very generic way to import files from your requirements to your project.

This method can be used for packaging applications and copying the result executables to your bin folder, or for copying assets, images, sounds, test static files, etc. Conan is a generic solution for package management, not only (but focused in) for C/C++ or libraries.

See also:

Check the section *Howtos/Manage shared libraries* to know more about working with shared libraries.

5.2 Using profiles

So far we have used the default settings stored in `~/.conan/profiles/default` and defined as command line arguments.

However, configurations can be large, settings can be very different, and we might want to switch easily between different configurations with different settings, options, etc. The best way to do it is using profiles.

A profile file contains a predefined set of `settings`, `options`, `environment variables`, and `build_requires` and has this structure:

```
[settings]
setting=value
```

(continues on next page)

(continued from previous page)

```
[options]
MyLib:shared=True

[env]
env_var=value

[build_requires]
Tool1/0.1@user/channel
Tool2/0.1@user/channel, Tool3/0.1@user/channel
*: Tool4/0.1@user/channel
```

Options allow definition with wildcards, to apply same option value to many packages:

```
[options]
*:shared=True
```

They would contain the desired configuration, for example:

Listing 1: *clang_3.5*

```
[settings]
os=Macos
arch=x86_64
compiler=clang
compiler.version=3.5
compiler.libcxx=libstdc++11
build_type=Release

[env]
CC=/usr/bin/clang
CXX=/usr/bin/clang++
```

You can store them in the default profile folder or anywhere in you project and you can use it instead of command line arguments:

```
$ conan create demo/testing -pr=clang_3.5
```

If we continue with the example of Poco, we could have a handy profile to help us build our project with the desired configuration and avoid the usage of all the command line arguments when installing the dependency packages.

A profile to install dependencies as **shared** and in **debug** mode will look like this:

Listing 2: *debug_shared*

```
include(default)

[settings]
build_type=Debug

[options]
Poco:shared=True
Poco:enable_apacheconnector=False
OpenSSL:shared=True
```

With this we could just install using the profile:

```
$ conan install . -pr=debug_shared
```

We could also create a new profile to use a different compiler version and store it in our project directory:

Listing 3: *poco_clang_3.5*

```
include(clang_3.5)

[options]
Poco:shared=True
Poco:enable_apacheconnector=False
OpenSSL:shared=True
```

Installation will be as easy as:

```
$ conan install . -pr=./poco_clang_3.5
```

See also:

Read more about *Profiles* for full reference.

5.3 Workflows

This section summarizes some possible layouts and workflows while using conan together with other tools as an end-user, i.e. installing and consuming existing packages. For creating your own packages, have a look at the *Packaging section*.

In both cases, the recommended approach is to have a conanfile (either .py or .txt) at the root of your project.

5.3.1 Single configuration

The single configuration is simple. It is the one that has been used so far for the examples and tutorials. In *Getting started*, we ran the **`conan install ..`** command inside the *build* folder and the *conaninfo.txt* and *conanbuildinfo.cmake* files were generated there. The build folder is temporary, you should exclude it from version control, so those temporary files are excluded too.

Out-of-source builds are also supported. Let's make a simple example:

```
$ git clone https://github.com/memsharded/example-hello.git
$ conan install ./example-hello --build=missing --install-folder example-hello-build
```

So the layout will be:

```
example-hello-build
  conaninfo.txt
  conanbuildinfo.txt
  conanbuildinfo.cmake
example-hello
  conanfile.txt
  CMakeLists.txt  # If using cmake, but can be Makefile, sln...
  main.cpp
```

Now you are ready to build:

```
$ cmake ../example-hello -G "Visual Studio 14 Win64"  # or other generator
$ cmake --build . --config Release
$ ./bin/greet
```

We have created a separate build configuration of the project, without affecting at all the original source directory. The benefit is that we can experiment freely, and even erase it and create a new build with a new configuration with different settings, if needed:

```
$ cd example-hello-build && rm -rf *
$ conan install ../example-hello -s compiler="<other compiler>" --build=missing
$ cmake ../example-hello -G "<other generator>"
$ cmake --build . --config Release
```

5.3.2 Multi configuration

You can also manage different configurations, in-source or out of source, and you can switch between them without taking the extra step of re-issuing the **conan install** command (even though this is not a speed-related issue, since the second time **conan install** is executed with the same parameters, it will run very fast: packages are installed in the local cache, not inside the project).

```
$ git clone https://github.com/memsharded/example-hello.git
$ conan install ./example-hello -s build_type=Debug --build=missing -if example-hello-
↪build/debug
$ conan install ./example-hello -s build_type=Release --build=missing -if example-
↪hello-build/release

$ cd example-hello-build/debug && cmake ../../example-hello -G "Visual Studio 14 Win64
↪" && cd ../..
$ cd example-hello-build/release && cmake ../../example-hello -G "Visual Studio 14␣
↪Win64" && cd ../..
```

Note: You can use the `--install-folder` or `-if` to specify where to generate the output files or create manually the directory and change to it before execute the **conan install** command.

So the layout will be:

```
example-hello-build
  debug
      conaninfo.txt
      conanbuildinfo.txt
      conanbuildinfo.cmake
      CMakeCache.txt # and other cmake files
  release
      conaninfo.txt
      conanbuildinfo.txt
      conanbuildinfo.cmake
      CMakeCache.txt # and other cmake files
example-hello
  conanfile.txt
  CMakeLists.txt  # If using cmake, but can be Makefile, sln...
  main.cpp
```

Now you can switch between your build configurations in exactly the same way you do for CMake or other build systems, moving to the folder in which the build configuration lives, because the conan configuration files for that

build configuration will also be there.

```
$ cd example-hello-build/debug && cmake --build . --config Debug && cd ../..
$ cd example-hello-build/release && cmake --build . --config Release && cd ../..
```

Note that the CMake `INCLUDE()` of your project must be prefixed with the current cmake binary directory, otherwise it will not find the necessary file:

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

CHAPTER

SIX

CREATING PACKAGES

This section shows how to create, build and test your packages.

6.1 Getting started

To start learning about creating packages, we will create a package from an existing source code repository: https://github.com/memsharded/hello. You can check that project, it is a very simple "hello world" C++ library, using CMake as build system to build a library and an executable. It has nothing related to conan in it.

We are using such github repository as an example, but the same process would apply to other source code origins, like downloading a zip or tarball from the internet.

Note: For this concrete example you will need, besides a C++ compiler, both *CMake* and *git* installed and in your path. They are not required by conan, you could use your own build system and version control instead.

6.1.1 Creating the package recipe

First, let's create a folder for our package recipe, and use the **conan new** helper command that will create a working package recipe for us:

```
$ mkdir mypkg && cd mypkg
$ conan new Hello/0.1 -t
```

This will generate the following files:

```
conanfile.py
test_package
  conanfile.py
  CMakeLists.txt
  example.cpp
```

At the root level, there is a *conanfile.py* which is the main recipe file, the one actually defining our package. Also there is a *test_package* folder, which contains a simple example consuming project that will require and link with the created package. It is useful to make sure that our package is correctly created.

Let's have a look to the root package recipe *conanfile.py*:

```
from conans import ConanFile, CMake, tools

class HelloConan(ConanFile):
```

(continues on next page)

(continued from previous page)

```
    name = "Hello"
    version = "0.1"
    settings = "os", "compiler", "build_type", "arch"
    options = {"shared": [True, False]}
    default_options = "shared=False"
    generators = "cmake"

    def source(self):
        self.run("git clone https://github.com/memsharded/hello.git")
        self.run("cd hello && git checkout static_shared")
        # This small hack might be useful to guarantee proper /MT /MD linkage in MSVC
        # if the packaged project doesn't have variables to set it properly
        tools.replace_in_file("hello/CMakeLists.txt", "PROJECT(MyHello)", ''
↪'PROJECT(MyHello)
include(${CMAKE_BINARY_DIR}/conanbuildinfo.cmake)
conan_basic_setup()''')

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

        # Explicit way:
        # self.run('cmake "%s/hello" %s' % (self.source_folder, cmake.command_line))
        # self.run("cmake --build . %s" % cmake.build_config)

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

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

This is a complete package recipe. Without worrying too much about every detail, these are the basics:

- The `settings` field defines the configuration that defines the different binary packages. In this example we are defining that any change to the OS, compiler, architecture or build type will generate a different binary package. Remember, Conan generates different binary packages for different introduced configuration (in this case settings) for the same recipe.

  Note that the platform where the recipe is running and the package is being build can be different from the final platform where the code will be running (`self.settings.os` and `self.settings.arch`) if the package is being cross-built. So if you want to apply a different build depending on the current build machine, you need to check it:

```
def build(self):
    if platform.system() == "Windows":
        cmake = CMake(self)
        cmake.configure(source_folder="hello")
        cmake.build()
    else:
        env_build = AutoToolsBuildEnvironment(self)
        env_build.configure()
```

(continues on next page)

(continued from previous page)

```
        env_build.make()
```

Learn more in the *Cross building* section.

- This package recipe is also able to create different binary packages for static and shared libraries with the `shared` option, which is defaulted to `False` (i.e. by default it will use static linkage).
- The `source()` method executes a **`git clone`** to retrieve the sources from github. Other origins, as downloading a zip file are also available. As you can see, any manipulation of the code can be done, as checking out any branch or tag, or patching the source code. In this example, we are adding two lines to the existing CMake code, to ensure binary compatibility. Don't worry too much about it now, we'll visit it later.
- The `build()` first configures the project, then builds it, with standard CMake commands. The `CMake` object is just a helper to ease the translation of conan settings to CMake command line arguments. Also remember that **CMake is not strictly required**. You can build packages directly invoking **make**, **MSBuild**, **SCons** or any other build system.

  See also:

  Check the *existing build helpers*.
- The `package()` method copies artifacts (headers, libs) from the build folder to the final package folder.
- Finally, the `package_info()` method defines that consumer must link with the "hello" library when using this package. Other information as include or lib paths can be defined as well. This information is used for files created by generators to be used by consumers, as *conanbuildinfo.cmake*.

6.1.2 The test_package folder

Note: The **test_package** is different from the library unit or integration tests, which should be more comprehensive. These tests are "package" tests, and validate that the package is properly created, and that package consumers will be able to link against it and reuse it.

If you have a look to the `test_package` folder, you will realize that the `example.cpp` and the `CMakeLists.txt` files don't have anything special. The *test_package/conanfile.py* file is just another recipe, you can think of it as the consumer *conanfile.txt* we have already seen in previous sections:

```
from conans import ConanFile, CMake
import os

class HelloTestConan(ConanFile):
    settings = "os", "compiler", "build_type", "arch"
    generators = "cmake"

    def build(self):
        cmake = CMake(self)
        cmake.configure()
        cmake.build()

    def imports(self):
        self.copy("*.dll", dst="bin", src="bin")
        self.copy("*.dylib*", dst="bin", src="lib")

    def test(self):
```

(continues on next page)

(continued from previous page)

```
        os.chdir("bin")
        self.run(".%sexample" % os.sep)
```

The main differences with the above *conanfile.py* are:

- It doesn't have a name and version, because we are not creating a package, so they are not necessary.
- The `package()` and `package_info()` methods are not required, since we are not creating a package.
- The `test()` method specifies which binaries have to be run.
- The `imports()` method is defined to copy shared libraries to the `bin` folder, so when dynamic linkage is used, and the `test()` method launches the `example` executable, they are found and `example` runs.

Note: An important difference with respect to normal package recipes, is that this one does not need to declare a `requires` attribute, to depend on the `Hello/0.1@demo/testing` package we are testing. This `requires` will be automatically injected by conan while running. You can however declare it explicitely, it will work, but you will have to remember to bump the version, and possibly the user and channel if you change them.

6.1.3 Creating and testing packages

We can create and test the package with our default settings simply by:

```
$ conan create . demo/testing
...
Hello world!
```

If you see "Hello world!", it worked.

This will perform the following steps:

- Copy ("export" in conan terms) the *conanfile.py* from the user folder into the **local cache**.
- Install the package, forcing building it from sources.
- Move to the *test_package* folder, and create a temporary *build* folder.
- Execute there a **`conan install ..`**, so it installs the requirements of the *test_package/conanfile.py*. Note that it will build "Hello" from sources.
- Build and launch the *example* consuming application, calling the *test_package/conanfile.py* `build()` and `test()` methods respectively.

Using conan commands, the **`conan create`** command would be equivalent to:

```
$ conan export . demo/testing
$ conan install Hello/0.1@demo/testing --build=Hello
# package is created now, use test to test it
$ conan test test_package Hello/0.1@demo/testing
```

The **`conan create`** command receives the same command line parameters as **`conan install`** so you can pass to it the same settings, options, and command line switches. If you want to create and test packages for different configurations, you could:

```
$ conan create . demo/testing -s build_type=Debug
$ conan create . demo/testing -o Hello:shared=True -s arch=x86
$ conan create . demo/testing -pr my_gcc49_debug_profile
```

(continues on next page)

(continued from previous page)

```
...
$ conan create ...
```

6.1.4 Settings vs. options

We have used settings as `os`, `arch` and `compiler`. But the above package recipe also contains a `shared` option (defined as `options = {"shared": [True, False]}`). What is the difference between settings and options?

Settings are project-wide configuration, something that typically affect to the whole project that is being built. For example the Operating System or the architecture would be naturally the same for all packages in a dependency graph, linking a Linux library for a Windows app, or mixing architectures is impossible.

Settings cannot be defaulted in a package recipe. A recipe for a given library cannot say that its default `os=Windows`. The `os` will be given by the environment in which that recipe is processed. It is a necessary input.

Settings are configurable. You can edit, add, remove settings or subsettings in your *settings.yml* file. See *the settings.yml reference*.

On the other hand, **options** are package-specific configuration. Being a static or shared library is not something that applies to all packages. Some can be header only libraries. Other packages can be just data, or package executables. Or packages can contain a mixture of different artifacts. `shared` is a common option, but packages can define and use any options they want.

Options are defined in the package recipe, including their allowed values, and it can be defaulted by the package recipe itself. A package for a library can well define that by default it will be a static library (a typical default). If no one else specifies something different, the package will be static.

There are some exceptions to the above, for example, settings can be defined per-package, like in command line:

```
$ conan install . -s MyPkg:compiler=gcc -s compiler=clang ..
```

This will use `gcc` for MyPkg and `clang` for the rest of the dependencies (extremely unusual case)

Or you can have a very widely used option in many packages and set its value all at once with patterns, like:

```
$ conan install . -o *:shared=True
```

Any doubts? Please check out our *FAQ section* or .

6.2 Recipe and sources in the same repo

In the previous package we implemented a `source()` method that fetched the source code from github. An alternative approach would be embedding the source code into the package recipe, so it is self-contained and it doesn't require to fetch code from external origins when it is necessary to build from sources.

This could be an appropriate approach if we want the package recipe to live in the same repository as the source code it is packaging. It could be considered as a "snapshot" of the source code too.

First, let's get the initial source code and create the basic package recipe:

```
$ conan new Hello/0.1 -t -s
```

A *src* folder will be created with the same "hello" source code than in the previous example. You can have a look at it, is straightforward code.

Now lets have a look to the *conanfile.py*:

```
from conans import ConanFile, CMake

class HelloConan(ConanFile):
    name = "Hello"
    version = "0.1"
    license = "<Put the package license here>"
    url = "<Package recipe repository url here, for issues about the package>"
    description = "<Description of Hello here>"
    settings = "os", "compiler", "build_type", "arch"
    options = {"shared": [True, False]}
    default_options = "shared=False"
    generators = "cmake"
    exports_sources = "src/*"

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

        # Explicit way:
        # self.run('cmake "%s/src" %s' % (self.source_folder, cmake.command_line))
        # self.run("cmake --build . %s" % cmake.build_config)

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

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

There are two important changes:

- Added the `exports_sources` field, to tell conan to copy all the files from the local *src* folder into the package recipe.
- Removed the `source()` method, it is not necessary anymore to retrieve external sources.

Also, you can notice the two CMake lines:

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

They are not added in the package recipe, as they can be directly put in the `src/CMakeLists.txt` file.

And simply create the package for user and channel **demo/testing** as previously:

```
$ conan create . demo/testing
...
Hello/0.1@demo/testing test package: Running test()
Hello world!
```

6.3 Package development flow

In the previous examples, we used **conan create** command to create a package of our library. Every time we run it, conan will perform some costly operations:

1. Copy the sources to a new and clean build folder.
2. Build the entire library from scratch.
3. Package the library once it is built.
4. Build the `test_package` example and test if it works.

But sometimes, specially with big libraries, while we are developing the recipe, **we cannot afford** to perform every time these operations.

The following section is the local development flow description based on the Bincrafters community blog.

The local workflow encourages users to do trial-and-error in a local sub-directory relative to their recipe, much like how developers typically test building their projects with other build tools. The strategy is to test the *conanfile.py* methods individually during this phase.

We will use the following conan flow example to follow the steps in the order below:

6.3.1 conan source

You will generally want to start off with the **conan source** command. The strategy here is that you're testing your source method in isolation, and downloading the files to a temporary sub-folder relative to the *conanfile.py*. This just makes it easier to get to the sources and validate them.

This method outputs the source files into the source-folder.

Input folders	Output folders
–	source-folder

```
$ cd example_conan_flow
$ conan source . --source-folder=tmp/source

PROJECT: Configuring sources in C:\Users\conan\example_conan_flow\tmp\source
Cloning into 'hello'...
...
```

Once you've got your source method right and it contains the files you expect, you can move on to testing the various attributes and methods relating to the downloading of dependencies.

6.3.2 conan install

Conan has multiple methods and attributes which relate to dependencies (all the ones with the word "require" in the name). The command **conan install** activates all them.

Input folders	Output folders
–	install-folder

```
$ conan install . --install-folder=tmp/install [--profile XXXX]

PROJECT: Installing C:\Users\conan\example_conan_flow\conanfile.py
Requirements
Packages
...
```

This also generates *conaninfo.txt* and *conanbuildinfo.xyz* (extension depends on generator you've used) in the temp folder (`install-folder`), which will be needed for the next step. Once you've got this command working with no errors, you can move on to testing the `build()` method.

6.3.3 conan build

The build method takes a path to a folder that has sources and also to the install folder to get the information of the settings and dependencies. It uses a path to a folder where it will perform the build.

Input folders	Output folders
`source-folder` `install-folder`	`build-folder`

```
$ conan build . --source-folder=tmp/source --install-folder=tmp/install --build-
↪folder=tmp/build

Project: Running build()
...
Build succeeded.
    0 Warning(s)
    0 Error(s)

Time Elapsed 00:00:03.34
```

This is pretty strightforward, but it does add a very helpful new shortcut for people who are packaging their own library. Now, developers can make changes in their normal source directory and just pass that path as the `--source-folder`.

6.3.4 conan package

Just as it sounds, this command now simply runs the `package()` method of a recipe. It needs all the information of the other folders in order to collect the needed information for the package: header files from source folder, settings and depency information from the install folder and built artifacts from the build folder.

Input folders	Output folders

`source-folder`
`install-folder`
`build-folder`
	`package-folder`

```
$ conan package . --source-folder=tmp/source --install-folder=tmp/install --build-
↪folder=tmp/build --package-folder=tmp/package

PROJECT: Generating the package
PROJECT: Package folder C:\Users\conan\example_conan_flow\tmp\package
```

(continues on next page)

(continued from previous page)

```
PROJECT: Calling package()
PROJECT package(): Copied 1 '.h' files: hello.h
PROJECT package(): Copied 2 '.lib' files: greet.lib, hello.lib
PROJECT: Package 'package' created
```

6.3.5 conan export-pkg

When you have checked that the packaged is done correctly, you can generate the package in the local cache. Note that the package is generated again to make sure this step is always reproducible.

This parameters takes the same parameters as `package()`.

Input folders	Output folders
source-folder install-folder build-folder package-folder	–

There are 2 modes of operation:

- Using `source-folder` and `build-folder``will use the ``package()` method to extract the artifacts from those folders and create the package, directly in the conan local cache. Strictly speaking, it doesn't require executing a `$ conan package` before, as it packages directly from those source and build folder, though `$ conan package` is still recommended in the dev-flow to debug the `package()` method.
- Using the `package-folder` argument (incompatible with the above 2), will not use the `package()` method, it will do an exact copy of the provided folder. It assumes the package has already been created by a previous `$ conan package` command or with a `$ conan build` command with a `build()` method running a `cmake.install()`.

```
$ conan export-pkg . user/testing --source-folder=tmp/source --install-folder=tmp/
↪install --build-folder=tmp/build

Packaging to 6cc50b139b9c3d27b3e9042d5f5372d327b3a9f7
Hello/0.1@user/channel: Generating the package
Hello/0.1@user/channel: Package folder C:\Users\conan\.conan\data\Hello\0.
↪1\user\channel\package\6cc50b139b9c3d27b3e9042d5f5372d327b3a9f7
Hello/0.1@user/channel: Calling package()
Hello/0.1@user/channel package(): Copied 2 '.lib' files: greet.lib, hello.lib
Hello/0.1@user/channel package(): Copied 2 '.lib' files: greet.lib, hello.lib
Hello/0.1@user/channel: Package '6cc50b139b9c3d27b3e9042d5f5372d327b3a9f7' created
```

6.3.6 conan test

The finally step to test the package for consumer is the test command. This step is quite straight-forward:

```
$ conan test test_package Hello/0.1@user/channel

Hello/0.1@user/channel (test package): Installing C:\Users\conan\repos\example_conan_
↪flow\test_package\conanfile.py
Requirements
    Hello/0.1@user/channel from local
```

(continues on next page)

(continued from previous page)

```
Packages
    Hello/0.1@user/channel:6cc50b139b9c3d27b3e9042d5f5372d327b3a9f7

Hello/0.1@user/channel: Already installed!
Hello/0.1@user/channel (test package): Generator cmake created conanbuildinfo.cmake
Hello/0.1@user/channel (test package): Generator txt created conanbuildinfo.txt
Hello/0.1@user/channel (test package): Generated conaninfo.txt
Hello/0.1@user/channel (test package): Running build()
...
```

There is often a need to repeatedly re-run the test to check the package is well generated for consumers.

As a summary, you could use the default folders and the flow would be as simple as:

```
$ git clone git@github.com:memsharded/example_conan_flow.git
$ cd example_conan_flow
$ conan source .
$ conan install .
$ conan build .
$ conan package .
...
PROJECT package(): Copied 1 '.h' files: hello.h
PROJECT package(): Copied 2 '.lib' files: greet.lib, hello.lib
PROJECT: Package 'package' created
```

6.3.7 conan create

Now we know we have all the steps of a recipe working. Thus, now is an appropriate time to try to run the recipe all the way through, and put it completely in the local cache.

The usual command for this is **`conan create`** and it basically performs the previous commands with **`conan test`** for the *test_package* folder:

```
$ conan create . user/channel
```

Even with this command, the package creator can iterate over the local cache if something does not work. This could be done with `--keep-source` and `--keep-build` flags.

If you see in the traces that the `source()` method has been properly executed but the package creation finally failed, you can skip the `source()` method the next time you issue **`conan create`** using **`--keep-source`**:

```
$ conan create . user/channel --keep-source

Hello/0.1@user/channel: A new conanfile.py version was exported
Hello/0.1@user/channel: Folder: C:\Users\conan\.conan\data\Hello\0.
↪1\user\channel\export
Hello/0.1@user/channel (test package): Installing C:\Users\conan\repos\example_conan_
↪flow\test_package\conanfile.py
Requirements
    Hello/0.1@user/channel from local
Packages
    Hello/0.1@user/channel:6cc50b139b9c3d27b3e9042d5f5372d327b3a9f7

Hello/0.1@user/channel: WARN: Forced build from source
Hello/0.1@user/channel: Building your package in C:\Users\conan\.conan\data\Hello\0.
↪1\user\channel\build\6cc50b139b9c3d27b3e9042d5f5372d327b3a9f7
```

(continues on next page)

(continued from previous page)

```
Hello/0.1@user/channel: Configuring sources in C:\Users\conan\.conan\data\Hello\0.
↪1\user\channel\source
Cloning into 'hello'...
remote: Counting objects: 17, done.
remote: Total 17 (delta 0), reused 0 (delta 0), pack-reused 17
Unpacking objects: 100% (17/17), done.
Switched to a new branch 'static_shared'
Branch 'static_shared' set up to track remote branch 'static_shared' from 'origin'.
Hello/0.1@user/channel: Copying sources to build folder
Hello/0.1@user/channel: Generator cmake created conanbuildinfo.cmake
Hello/0.1@user/channel: Calling build()
...
```

If you see that library builds correctly too, you can do the same to skip also the `build()` step with the `--keep-build` flag:

```
$ conan create --keep-build
```

6.4 Packaging existing binaries

Sometimes, it is necessary to create packages from existing binaries, like binaries from third parties, or previously built by another process or team not using conan, so building from sources is not wanted. You would want to package local files in two situations:

- When it is not possible to build the packages from sources (only pre-built binaries available).
- When you are developing your package locally and want to export the built artifacts to the local cache. As you don't want to rebuild again (clean copy) your artifacts, you don't want to call **`conan create`**. This way you can keep your build cache if you are using an IDE or calling locally to the **`conan build`** command.

6.4.1 Packaging pre-built binaries

If the files we want to package are just local, creating a `build()` method that would copy them from the user folder is not reproducible, so it doesn't add any value. For this use case, it is possible to use **`conan export-pkg`** command directly.

A conan recipe is still needed, in this case it will be very simple, just the meta information of the package. A basic recipe can be created with the **`conan new`** command:

```
$ conan new Hello/0.1 --bare
```

This will create and store in the local cache the following package recipe:

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

    def package(self):
        self.copy("*")

    def package_info(self):
        self.cpp_info.libs = self.collect_libs()
```

The provided `package_info()` method will scan the package files to provide the end consumers with the name of the libraries to link with. This method can be further customized to provide other build flags (typically conditioned to the settings). The default `package_info()` applies: it will define headers in "include" folder, libraries in "lib" folder, binaries in "bin" folder. A different package layout can be defined in `package_info()` method.

This package recipe can be also extended to provide support for more configurations (for example, adding options: shared/static, or using different settings), adding dependencies (`requires`), etc.

Then, we will assume that we have in our current directory a *lib* folder with some binary for this "hello" library *libhello.a*, compatible for example with Windows MinGW (gcc) version 4.9:

```
$ conan export-pkg . Hello/0.1@myuser/testing  -s os=Windows -s compiler=gcc -s␣
↪compiler.version=4.9 ...
```

Having a *test_package* folder is still very recommended, to locally test the package before uploading. As we don't want to build the package from sources, the flow would be:

```
$ conan new Hello/0.1 --bare --test
# customize test_package project
# customize package recipe if necessary
$ cd my/path/to/binaries
$ conan export-pkg PATH/TO/conanfile.py Hello/0.1@myuser/testing  -s os=Windows -s␣
↪compiler=gcc -s compiler.version=4.9 ...
$ conan test PATH/TO/test_package/conanfile.py Hello/0.1@myuser/testing -s os=Windows␣
↪-s compiler=gcc -s ...
```

The last 2 steps can be repeated for any number of configurations.

6.4.2 Downloading and Packaging pre-built binaries

In this case, having a complete conan recipe, with the detailed retrieval of the binaries could be the preferred way, because it has better reproducibility, and the original binaries might be traced. Such a recipe would be like:

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

    def build(self):
        if self.settings.os == "Windows" and self.compiler == "Visual Studio":
            url = ("https://<someurl>/downloads/hello_binary%s_%s.zip"
                   % (str(self.settings.compiler.version), str(self.settings.build_
↪type)))
        elif ...:
            url = ...
        else:
            raise Exception("Binary does not exist for these settings")
        tools.get(url)

    def package(self):
        self.copy("*") # assume package as-is, but you can also copy specific files␣
↪or rearrange

    def package_info(self):  # still very useful for package consumers
        self.cpp_info.libs = ["hello"]
```

Typically, pre-compiled binaries come for different configurations, so the only task that the `build()` method has to implement is to map the `settings` to the different URLs.

Note:

- This is a normal conan package, even if the binaries are being retrieved from somewhere. The **recommended approach** is using **`conan create`**, and have a small consuming project besides the above recipe, to test locally, then upload the conan package with the binaries to the conan remote with **`conan upload`**.
- The same building policies apply. Having a recipe will fail if no conan packages are created, and the **`--build`** argument is not defined. A typical approach for this kind of packages could be to define a **`build_policy="missing"`**, especially if the URLs are also under the team control. If they are external (internet), it could be better to create the packages and store them in your own conan server, so builds do not rely on the third party URL being available.

6.5 Understanding packaging

6.5.1 Manual package creation and testing

The previous **create** approach using *test_package* subfolder, is not strictly necessary, though **very strongly recommended**. If we didn't want to use the *test_package* functionality, we could just write our recipe ourselves or use the **`conan new`** command without the **`-t`**. command line argument.

```
$ mkdir mypkg && cd mypkg
$ conan new Hello/0.1
```

This will create just the *conanfile.py* recipe file. Now we could create our package:

```
$ conan create demo/testing
```

This would be equivalent to:

```
$ conan export . demo/testing
$ conan install Hello/0.1@demo/testing --build=Hello
```

Once the package is there, it can be consumed like any other package, just add `Hello/0.1@demo/testing` to some project *conanfile.txt* or *conanfile.py* requirements and run:

```
$ conan install .
# build and run your project to ensure the package works
```

6.5.2 The package creation process

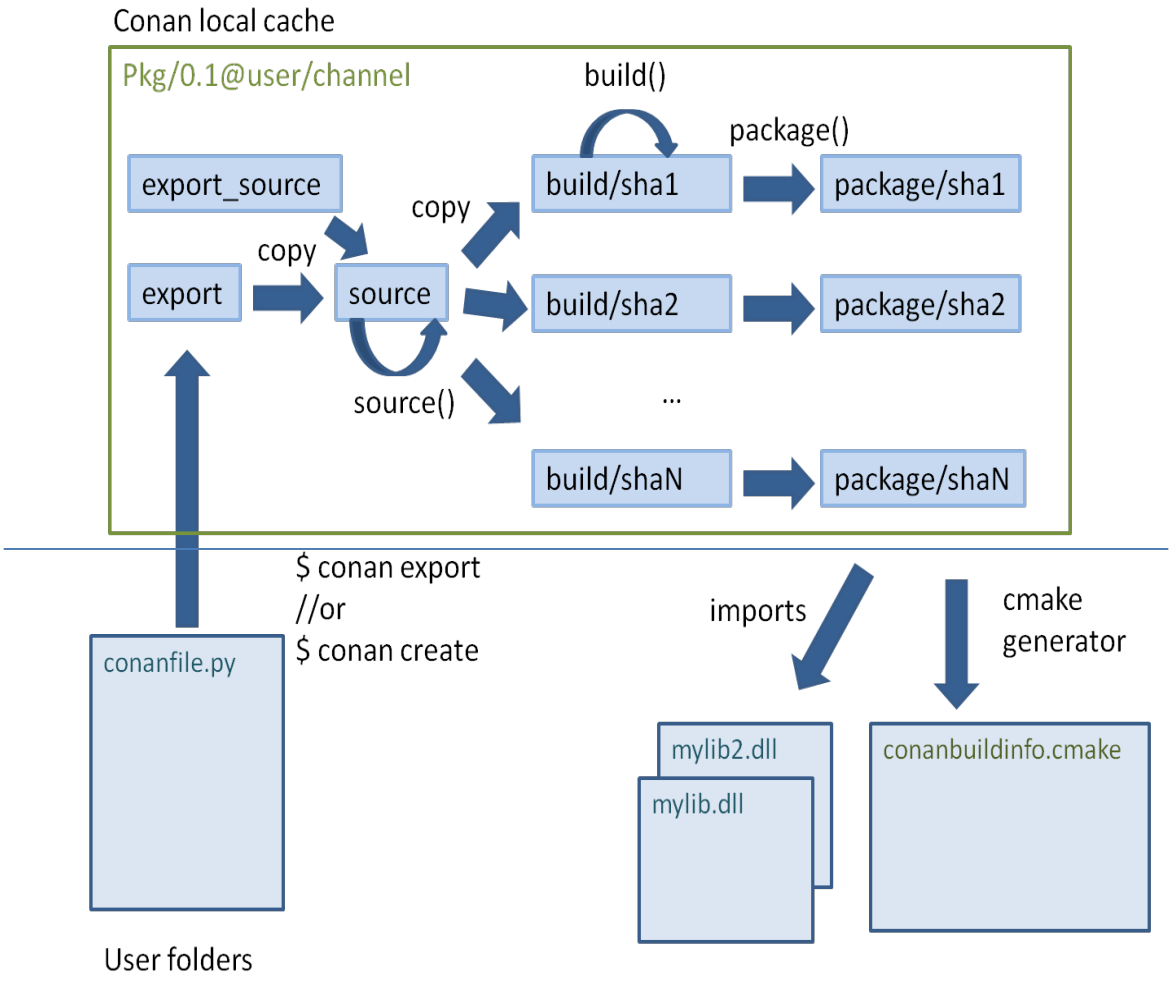
It is very useful for package creators and conan users in general to understand the flow of package creation inside the conan local cache, and its layout.

For every package recipe, there are 5 important folders in the **local cache**:

- **export**: The folder where the package recipe is stored.
- **export_source**: The folder where code copied with the recipe `exports_sources` attribute is stored.
- **source**: Where the source code for building from sources is stored.
- **build**: Where the actual compilation of sources is done. There will typically be one subfolder for each different binary configuration

- **package**: Where the final package artifacts are stored. There will be one subfolder for each different binary configuration

The *source* and *build* folders only exist when the packages have been built from sources.



The process starts when a package is "exported", via the **`conan export`** command or more typically, with the **`conan create`** command. The *conanfile.py* and files specified by the `exports_sources` field are copied from the user space into the **local cache**.

The *export* and *export_source* files are copied to the *source* folder, and then the `source()` method is executed (if existing). Note that there is only one source folder for all the binary packages. If some source code is to be generated that will be different for different configurations, it cannot be generated in the `source()` method, it has to be done in the `build()` method.

Then, for each different configuration of settings and options, a package ID will be computed in the form of a SHA-1 hash of such configuration. Sources will be copied to the *build/hashXXX* folder, and the `build()` method will be triggered.

After that, the `package()` method will be called to copy artifacts from the *build/hashXXX* folder to the *package/hashXXX* folder.

Finally, the `package_info()` methods of all dependencies will be called and gathered to be able to generate files for the consumer build system, as the *conanbuildinfo.cmake* for the `cmake` generator. Also the `imports` feature will copy artifacts from the local cache into user space if specified.

Any doubts? Please check out our *FAQ section* or .

6.6 Define package ABI compatibility

Each package recipe can generate *N* binary packages from it, depending on three things: `settings`, `options` and `requires`.

When any of the *settings* of a package recipe changes, it will reference a different binary:

```
class MyLibConanPackage(ConanFile):
    name = "MyLib"
    version = "1.0"
    settings = "os", "arch", "compiler", "build_type"
```

When this package is installed by a *conanfile.txt*, another package *conanfile.py*, or directly:

```
$ conan install MyLib/1.0@user/channel -s arch=x86_64 -s ...
```

The process will be:

1. Conan will get the user input settings and options, which can come from the command line, be default values defined in *~/.conan/profiles/default*, defined in a *Profiles* file, or cached from the latest **`conan install`** execution.
2. Conan will retrieve the `MyLib/1.0@user/channel` recipe, read the `settings` attribute, and assign the necessary values.
3. With the current package values for `settings` (also `options` and `requires`), it will compute a SHA1 hash, that will be the binary package ID, e.g. `c6d75a933080ca17eb7f076813e7fb21aaa740f2`.
4. Conan will try to find the `c6d75...` binary package. If it's present conan will retrieve it, if not, it can be built from sources with **`conan install --build`**.

If the package is installed again with different settings, for example, for 32bits architecture:

```
$ conan install MyLib/1.0@user/channel -s arch=x86 -s ...
```

The process will be repeated, but now generating a different package ID, because the `arch` setting will have a different value. The same applies for different compilers, compiler versions, build type, etc., generating multiple binaries, one for each configuration.

When users of the package define the same settings as one of those binaries that have been uploaded, the computed package ID will be the same, such binary will be retrieved, and they will be able to reuse the binary without building it from sources.

The use case for `options` is very similar, the main difference is that options can be more easily defined at the package level and they can be defaulted. Check the *options, default_options* reference.

Note the simple scenario of a **header-only** library. Such package does not need to be built, and it will not have any ABI issues at all. The recipe of such package will have to generate exactly 1 binary package, no more. This is easily achieved, just by no declaring `settings` nor `options` in the recipe:

```
class MyLibConanPackage(ConanFile):
    name = "MyLib"
    version = "1.0"
    # no settings defined!
```

Then, no matter what are the settings defined by the users, which compiler or version, the package settings and options will always be the same (empty) and they will hash to the same binary package ID, that will typically contain just the header files.

What happens if we have a library that we know we can build with gcc 4.8 and we know it will remain ABI compatibility with gcc 4.9? This kind of compatibility is easier to achieve for example for pure C libraries. Although it could be argued that it is worth rebuilding with 4.9 too, to get fixes and performance improvements, lets suppose that we don't want to create 2 different binaries, just one built with gcc 4.8 and be able to use it from gcc 4.9 installs.

6.6.1 Defining a custom package_id()

The default `package_id()` uses the `settings` and `options` directly as defined, and assumes semver behavior for dependencies `requires`.

This `package_id()` recipe method can be overriden to control the package ID generation. Within the `package_id()` method we have access to the `self.info` object, which is the actual object being hashed for computing the binary ID:

- **self.info.settings**: Contains all the declared settings, always as string values. We can access/alter the settings. E.g: `self.info.settings.compiler.version`.
- **self.info.options**: Contains all the declared options, always as string values. E.g: `self.info.options.shared`.

Initially, this `info` object will contain the original settings and options, stored as strings. They can be changed without constraints, to any other string value.

For example, if you are sure your package ABI compatibility is fine for GCC versions > 4.5 and < 5.0, (just an example, not a real case) you could do this:

```
from conans import ConanFile, CMake, tools
from conans.model.version import Version

class PkgConan(ConanFile):
    name = "Pkg"
    version = "0.1"
    settings = "compiler", "build_type"

    def package_id(self):
        v = Version(str(self.settings.compiler.version))
        if self.settings.compiler == "gcc" and (v >= "4.5" and v < "5.0"):
            self.info.settings.compiler.version = "GCC 4 between 4.5 and 5.0"
```

We have set the `self.info.settings.compiler.version` with an arbitrary string, it's not really important, could be any string. The only important thing is that won't change for any GCC[4.5-5.0], for those gcc versions, it will be always the same string, and then it will be always hashed to the same ID.

Let's check that it works properly installing the package for gcc 4.5:

```
$ conan export myuser/mychannel
$ conan install Pkg/1.0@myuser/mychannel -s compiler=gcc -s compiler.version=4.5 ...

Requirements
    Pkg/1.0@myuser/mychannel from local
Packages
    Pkg/1.0@myuser/mychannel:mychannel:af044f9619574eceb8e1cca737a64bdad88246ad
...
```

We can see that the computed package ID is `af04...46ad` (not real). What would happen if we specify GCC 4.6?

```
$ conan install Pkg/1.0@myuser/mychannel -s compiler=gcc -s compiler.version=4.6 ...

Requirements
    Pkg/1.0@myuser/mychannel from local
Packages
    Pkg/1.0@myuser/mychannel:mychannel:af044f9619574eceb8e1cca737a64bdad88246ad
```

Same result, the required package is again `af04...46ad`. Now we can try with GCC 4.4 (<4.5).

```
$ conan install Pkg/1.0@myuser/mychannel -s compiler=gcc -s compiler.version=4.4 ...

Requirements
    Pkg/1.0@myuser/mychannel from local
Packages
    Pkg/1.0@myuser/mychannel:mychannel:7d02dc01581029782b59dcc8c9783a73ab3c22dd
```

Now the computed package ID is different, that means that we need a different binary package for GCC 4.4.

The same way we have adjusted the `self.info.settings` we could set the `self.info.options` values if necessary.

See also:

Check the *package_id() method reference* too see the available helper methods to change the package_id() behavior, for example to:

- Adjust our package recipe as a **header only**
- Adjust **Visual Studio toolsets** compatibility

6.6.2 The problem of dependencies

Let's define a simple scenario in which there are two packages, one for `MyLib/1.0` which depends on (requires) `MyOtherLib/2.0`. The recipes and binaries for them have been created and uploaded to a conan server.

A new release for `MyOtherLib/2.1` comes out, with improved recipe and new binaries. The `MyLib/1.0` is modified to upgrade the requires to `MyOtherLib/2.1`. (Note that this is not strictly necessary, we would face the same problem if the downstream, consuming project defines a dependency to `MyOtherLib/2.1`, which would have precedence over the existing one in MyLib).

The question is: **Is it necessary to build new MyLib/1.0 packages binaries?** Or the existing packages are still valid?

The answer: **It depends**.

Let's suppose that both are being compiled as static libraries, and that the API exposed by `MyOtherLib` to `MyLib/1.0` through the public headers has not changed at all. Then, it is not necessary to build new binaries for `MyLib/1.0`, because the final consumer will link against both `Mylib/1.0` and `MyOtherLib/2.1`.

It could happen that the API exposed by **MyOtherLib** in public headers has changed, but without affecting the `MyLib/1.0` binary, for whatever reasons, like changes consisting on new functions, not used by **MyLib**. The same reasoning would still be valid if **MyOtherLib** was header only.

But what if one header file of `MyOtherLib`, named *myadd.h* has changed from `2.0`:

```
int addition (int a, int b) { return a - b; }
```

To the *myadd.h* file in `2.1`:

```
int addition (int a, int b) { return a + b; }
```

And the `addition()` function is being called from compiled `.cpp` files of `MyLib/1.0`?

Then, in this case, **MyLib/0.1 has to build a new binary for the new dependency version**. Otherwise, it will maintain the old, buggy `addition()` version. Even if `MyLib/0.1` hasn't change a line, not the code, neither the recipe, still the resulting binary would be different.

6.6.3 Using package_id() for package dependencies

The `self.info` object also have a `requires` object. It is a dictionary with the necessary information for each requirement, all direct and transitive dependencies. E.g. `self.info.requires["MyOtherLib"]` is a `RequirementInfo` object.

- Each `RequirementInfo` has the following *read only* reference fields:
  - `full_name`: Full require's name. E.g **MyOtherLib**
  - `full_version`: Full require's version. E.g **1.2**
  - `full_user`: Full require's user. E.g **my_user**
  - `full_channel`: Full require's channel. E.g **stable**
  - `full_package_id`: Full require's package ID. E.g **c6d75a...**
- The following fields are the ones used in the `package_id()` evaluation:
  - `name`: By default same value as full_name. E.g **MyOtherLib**.
  - `version`: By default the major version representation of the `full_version`. E.g **1.Y** for a **1.2** `full_version` field and **1.Y.Z** for a **1.2.3** `full_version` field.
  - `user`: By default `None` (doesn't affect the package ID).
  - `channel`: By default `None` (doesn't affect the package ID).
  - `package_id`: By default `None` (doesn't affect the package ID).

When defining a package ID to model dependencies, it is necessary to take into account two factors:

- The versioning schema followed by our requirements (semver?, custom?)
- Type of library being built and type of library being reused (shared: so, dll, dylib, static).

Versioning schema

By default conan assumes **semver** compatibility, i.e, if a version changes from minor **2.0** to **2.1** conan will assume that the API is compatible (headers not changing), and that it is not necessary to build a new binary for it. Exactly the same for patches, changing from **2.1.10** to **2.1.11** doesn't require a re-build. Those rules are defined by semver.

If it is necessary to change the default behavior, the applied versioning schema can be customized within the `package_id()` method:

```
from conans import ConanFile, CMake, tools
from conans.model.version import Version

class PkgConan(ConanFile):
    name = "Mylib"
    version = "1.0"
    settings = "os", "compiler", "build_type", "arch"
    requires = "MyOtherLib/2.0@lasote/stable"
```

(continues on next page)

(continued from previous page)

```
    def package_id(self):
        myotherlib = self.info.requires["MyOtherLib"]

        # Any change in the MyOtherLib version will change current Package ID
        myotherlib.version = myotherlib.full_version

        # Changes in major and stable versions will change the Package ID but
        # only a MyOtherLib revision won't. E.j: From 1.2.3 to 1.2.89 won't change.
        myotherlib.version = myotherlib.full_version.minor()
```

Besides the `version`, there are some other helpers that can be used, to decide whether the **channel** and **user** of one dependency also affects the binary package, or even the required package ID can change your own package ID:

```
def package_id(self):
    # Default behavior, only major release changes the package ID
    self.info.requires["MyOtherLib"].semver_mode()

    # Any change in the require version will change the package ID
    self.info.requires["MyOtherLib"].full_version_mode()

    # Any change in the MyOtherLib version, user or channel will affect our package ID
    self.info.requires["MyOtherLib"].full_recipe_mode()

    # Any change in the MyOtherLib version, user or channel or Package ID will affect␣
↪our package ID
    self.info.requires["MyOtherLib"].full_package_mode()

    # The requires won't affect at all to the package ID
    self.info.requires["MyOtherLib"].unrelated_mode()
```

You can also adjust the individual properties manually:

```
def package_id(self):
    myotherlib = self.info.requires["MyOtherLib"]

    # Same as myotherlib.semver_mode()
    myotherlib.name = myotherlib.full_name
    myotherlib.version = myotherlib.full_version.stable()
    myotherlib.user = myotherlib.channel = myotherlib.package_id = None

    # Only the channel (and the name) matters
    myotherlib.name = myotherlib.full_name
    myotherlib.user = myotherlib.package_id = myotherlib.version = None
    myotherlib.channel = myotherlib.full_channel
```

The result of the `package_id()` is the package ID hash, but the details can be checked in the generated *conaninfo.txt* file. The `[requires]`, `[options]` and `[settings]` are those taken into account to generate the SHA1 hash for the package ID, while the [full_xxxx] fields show the complete reference information.

The default behavior produces a *conaninfo.txt* that looks like:

```
[requires]
  MyOtherLib/2.Y.Z

[full_requires]
  MyOtherLib/2.2@demo/testing:73bce3fd7eb82b2eabc19fe11317d37da81afa56
```

Library types: Shared, static, header only

Let's see some examples, corresponding to common scenarios:

- `MyLib/1.0` is a shared library that links with a static library `MyOtherLib/2.0` package. When a new `MyOtherLib/2.1` version is released: Do I need to create a new binary for `MyLib/1.0` to link with it?

  Yes, always, because the implementation is embedded in the `MyLib/1.0` shared library. If we always want to rebuild our library, even if the channel changes (we assume a channel change could mean a source code change):

```
def package_id(self):
    # Any change in the MyOtherLib version, user or
    # channel or Package ID will affect our package ID
    self.info.requires["MyOtherLib"].full_package_mode()
```

- `MyLib/1.0` is a shared library, requiring another shared library `MyOtherLib/2.0` package. When a new `MyOtherLib/2.1` version is released: Do I need to create a new binary for `MyLib/1.0` to link with it?

  It depends, if the public headers have not changed at all, it is not necessary. Actually it might be necessary to consider transitive dependencies that are shared among the public headers, how they are linked and if they cross the frontiers of the API, it might also lead to incompatibilities. If public headers have changed, it would depend on what changes and how are they used in `MyLib/1.0`. Adding new methods to the public headers will have no impact, but changing the implementation of some functions that will be inlined when compiled from `MyLib/1.0` will definitely require re-building. For this case, it could make sense:

```
def package_id(self):
    # Any change in the MyOtherLib version, user or channel
    # or Package ID will affect our package ID
    self.info.requires["MyOtherLib"].full_package_mode()

    # Or any change in the MyOtherLib version, user or
    # channel will affect our package ID
    self.info.requires["MyOtherLib"].full_recipe_mode()
```

- `MyLib/1.0` is a header-only library, linking with any kind (header, static, shared) of library in `MyOtherLib/2.0` package. When a new `MyOtherLib/2.1` version is released: Do I need to create a new binary for `MyLib/1.0` to link with it?

  Never, the package should always be the same, there are no settings, no options, and in any way a dependency can affect a binary, because there is no such binary. The default behavior should be changed to:

```
def package_id(self):
    self.info.requires.clear()
```

- `MyLib/1.0` is a static library, linking with a header only library in `MyOtherLib/2.0` package. When a new `MyOtherLib/2.1` version is released: Do I need to create a new binary for `MyLib/1.0` to link with it? It could happen that the `MyOtherLib` headers are strictly used in some `MyLib` headers, which are not compiled, but transitively #included. But in the general case it is likely that `MyOtherLib` headers are used in `MyLib` implementation files, so every change in them should imply a new binary to be built. If we know that changes in the channel never imply a source code change, because it is the way we have defined our workflow/lifecycle, we could write:

```
def package_id(self):

    self.info.requires["MyOtherLib"].full_package()
    self.info.requires["MyOtherLib"].channel = None # Channel doesn't change out␣
↪package ID
```

6.7 Inspecting packages

You can inspect the uploaded packages and also the packages in the local cache with the **conan get** command.

- List the files of a local recipe folder:

```
$ conan get zlib/1.2.8@conan/stable .

Listing directory '.':
 CMakeLists.txt
 conanfile.py
 conanmanifest.txt
```

- Print the *conaninfo.txt* file of a binary package:

```
$ conan get zlib/1.2.11@conan/stable -p 09512ff863f37e98ed748eadd9c6df3e4ea424a8
```

- Print the *conanfile.py* from a remote package:

```
$ conan get zlib/1.2.8@conan/stable -r conan-center
```

```
from conans import ConanFile, tools, CMake, AutoToolsBuildEnvironment
from conans.util import files
from conans import __version__ as conan_version
import os


class ZlibConan(ConanFile):
    name = "zlib"
    version = "1.2.8"
    ZIP_FOLDER_NAME = "zlib-%s" % version

    #...
```

Check the *conan get command* command reference and more examples.

6.8 Packaging approaches

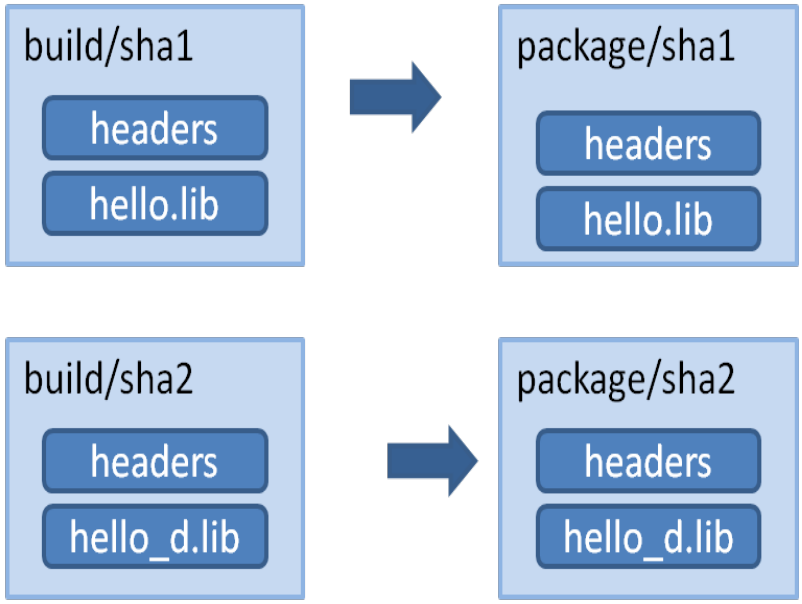
Package recipes have three methods to control the package's binary compatibility and to implement different packaging approaches: *package_id()*, *build_id()* and *package_info()*.

The above methods let package creators follow different package approaches to choose the best fit for each library.

6.8.1 1 config (1 build) -> 1 package

A typical approach is to have one configuration for each package containing the artifacts. In this approach, for example, the debug pre-compiled libraries will be in a different package than the release pre-compiled libraries.

So if there is a package recipe that builds a "hello" library, there will be one package containing the release version of the "hello.lib" library and a different package containing a debug version of that library (in the figure denoted as "hello_d.lib", to make it clear, it is not necessary to use different names).



In this approach, the `package_info()` method can just set the appropriate values for consumers, to let them know about the package library names, and necessary definitions and compile flags.

```
class HelloConan(ConanFile):

    settings = "os", "compiler", "build_type", "arch"

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

It is very important to note that it is declaring the `build_type` as a setting. This means that a different package will be generated for each different value of such setting.

The values that packages declare here (the *include*, *lib* and *bin* subfolders are already defined by default, so they define the include and library path to the package) are translated to variables of the respective build system by the used generators. That is, if using the `cmake` generator, such above definition will be translated in *conanbuildinfo.cmake* to something like:

```
set(CONAN_LIBS_MYPKG mylib)
# ...
set(CONAN_LIBS mylib ${CONAN_LIBS})
```

Those variables, will be used in the `conan_basic_setup()` macro to actually set cmake relevant variables.

If the developer wants to switch configuration of the dependencies, he will usually switch with:

```
$ conan install -s build_type=Release ...
# when need to debug
$ conan install -s build_type=Debug ...
```

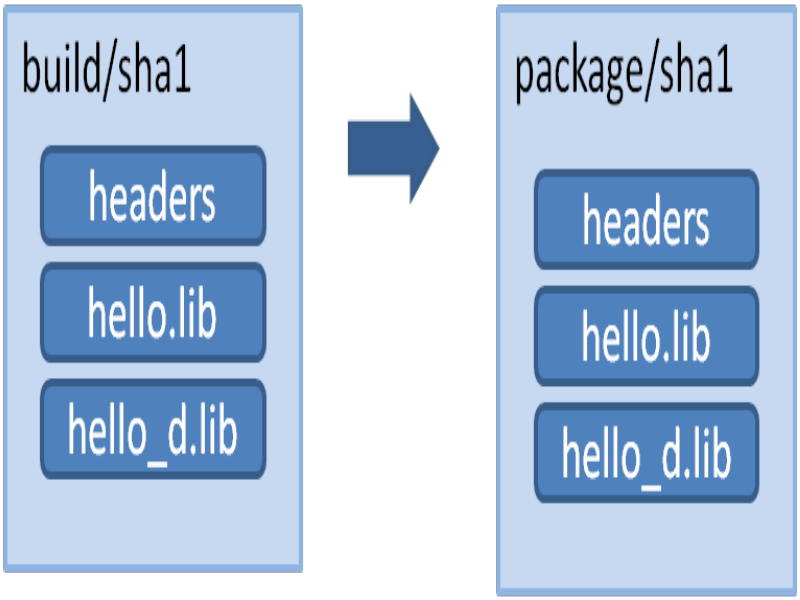
These switches will be fast, since all the dependencies are already cached locally.

This process has some advantages: it is quite easy to implement and maintain. The packages are of minimal size, so disk space and transfers are faster, and builds from sources are also kept to the necessary minimum. The decoupling of configurations might help with isolating issues related to mixing different types of artifacts, and also protecting valuable information from deploy and distribution mistakes. For example, debug artifacts might contain symbols or source code, which could help or directly provide means for reverse engineering. So distributing debug artifacts by mistake could be a very risky issue.

Read more about this in *package_info()*.

6.8.2 N configs -> 1 package

It is possible that someone wants to package both debug and release artifacts in the same package, so it can be consumed from IDEs like Visual Studio changing debug/release configuration from the IDE, and not having to specify it in the command line. This type of package will include different artifacts for different configurations, like both the release and debug version of the "hello" library, in the same package.



Note: A complete working example of the following code can be found in a github repo. You should be able to run:

```
$ git clone https://github.com/memsharded/hello_multi_config
$ cd hello_multi_config
$ conan create . user/channel -s build_type=Release
$ conan create . user/channel -s build_type=Debug --build=missing
```

Creating a multi-configuration Debug/Release package is not difficult, see the following example using CMake.

The first step is to remove `build_type` from the settings. It will not be an input setting, the generated package will always be the same, containing both Debug and Release artifacts. The Visual Studio runtime is different for debug and release (`MDd` or `MD`), so if we are fine with the default runtime (MD/MDd), it is also good to remove the `runtime` subsetting in the `configure()` method:

```
class Pkg(ConanFile):
    # build_type has been ommitted. It is not an input setting.
    settings = "os", "compiler", "arch"

    def configure(self):
        # it is also necessary to remove the VS runtime
        if self.settings.compiler == "Visual Studio":
            del self.settings.compiler.runtime

    def build(self):
```

(continues on next page)

(continued from previous page)

```
        cmake = CMake(self)
        if cmake.is_multi_configuration:
            cmmd = 'cmake "%s" %s' % (self.source_folder, cmake.command_line)
            self.run(cmmd)
            self.run("cmake --build . --config Debug")
            self.run("cmake --build . --config Release")
        else:
            for config in ("Debug", "Release"):
                self.output.info("Building %s" % config)
                self.run('cmake "%s" %s -DCMAKE_BUILD_TYPE=%s'
                         % (self.source_folder, cmake.command_line, config))
                self.run("cmake --build .")
                shutil.rmtree("CMakeFiles")
                os.remove("CMakeCache.txt")
```

In this case, we are assuming that the binaries will be differentiated with a suffix, in cmake syntax:

```
set_target_properties(mylibrary PROPERTIES DEBUG_POSTFIX _d)
```

Such a package can define its information for consumers as:

```
def package_info(self):
    self.cpp_info.release.libs = ["mylibrary"]
    self.cpp_info.debug.libs = ["mylibrary_d"]
```

This will translate to the cmake variables:

```
set(CONAN_LIBS_MYPKG_DEBUG mylibrary_d)
set(CONAN_LIBS_MYPKG_RELEASE mylibrary)
# ...
set(CONAN_LIBS_DEBUG mylibrary_d ${CONAN_LIBS_DEBUG})
set(CONAN_LIBS_RELEASE mylibrary ${CONAN_LIBS_RELEASE})
```

And these variables will be correctly applied to each configuration by `conan_basic_setup()` helper.

In this case you can still use the general, not config-specific variables. For example, the include directory, set by default to *include*, is still the same for both debug and release. Those general variables will be applied for all configurations.

Important: The above code assumes that the package will always use the default Visual Studio runtime (MD/MDd). If we want to keep the package configurable for supporting static(MT)/dynamic(MD) linking with the VS runtime library, some extra work is needed. Basically:

- Keep, the `compiler.runtime` setting, i.e. do not implement the `configure()` method removing it
- Don't let the `CMake` helper define the `CONAN_LINK_RUNTIME` env-var to define the runtime, because being defined by the consumer it would be incorrectly applied to both Debug and Release artifacts. This can be done with a `cmake.command_line.replace("CONAN_LINK_RUNTIME", "CONAN_LINK_RUNTIME_MULTI")` to define a new variable
- Write a `package_id()` method that defines the packages to be built, one for MD/MDd, and other for MT/MTd
- In *CMakeLists.txt*, use the `CONAN_LINK_RUNTIME_MULTI` variable to correctly setup up the runtime for debug and release flags

All these steps are already coded in the repo https://github.com/memsharded/hello_multi_config and commented out as **"Alternative 2"**

Also, you can use any custom configuration you want, they are not restricted. For example, if your package is a multi-library package, you could try doing something like:

```
def package_info(self):
    self.cpp_info.regex.libs = ["myregexlib1", "myregexlib2"]
    self.cpp_info.filesystem.libs = ["myfilesystemlib"]
```

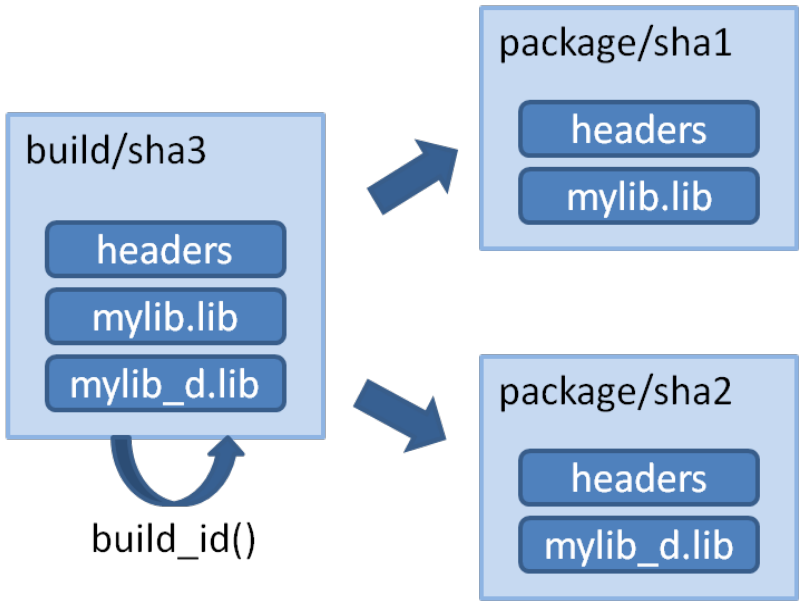
These specific config variables will not be automatically applied, but you can directly use them in your consumer CMake build script.

Note: The automatic conversion of multi-config variables to generators is currently only implemented in the `cmake` and `txt` generators. If you want to have support for them in another build system, please open a GitHub issue for it.

6.8.3 N configs (1 build) -> N packages

It's possible that an already existing build script is building binaries for different configurations at once, like debug/release, or different architectures (32/64bits), or library types (shared/static). If such build script is used in the previous "Single configuration packages" approach, it will definitely work without problems, but we'll be wasting precious build time, as we'll be re-building the whole project for each package, then extracting the relevant artifacts for the given configuration, leaving the others.

It is possible to specify the logic, so the same build can be reused to create different packages, which will be more efficient:



This can be done by defining a `build_id()` method in the package recipe that will specify the logic.

```
settings = "os", "compiler", "arch", "build_type"

def build_id(self):
    self.info_build.settings.build_type = "Any"

def package(self):
    if self.settings.build_type == "Debug":
        #package debug artifacts
```

(continues on next page)

(continued from previous page)

```
    else:
        # package release
```

Note that the `build_id()` method uses the `self.info_build` object to alter the build hash. If the method doesn't change it, the hash will match the package folder one. By setting `build_type="Any"`, we are forcing that for both Debug and Release values of `build_type`, the hash will be the same (the particular string is mostly irrelevant, as long as it is the same for both configurations). Note that the build hash `sha3` will be different of both `sha1` and `sha2` package identifiers.

This doesn't imply that there will be strictly one build folder. There will be a build folder for every configuration (architecture, compiler version, etc). So if we just have Debug/Release build types, and we're producing N packages for N different configurations, we'll have N/2 build folders, saving half of the build time.

Read more about this in *build_id()*.

6.9 Tools for package creators

With some python (or just pure shell or bash) scripting, we could easily automate the whole package creation and testing process, for many different configurations. For example you could put the following script in the package root folder. Name it *build.py*:

```
import os, sys
import platform

def system(command):
    retcode = os.system(command)
    if retcode != 0:
        raise Exception("Error while executing:\n\t %s" % command)

if __name__ == "__main__":
    params = " ".join(sys.argv[1:])

    if platform.system() == "Windows":
        system('conan create demo/testing -s compiler="Visual Studio" -s compiler.
↪version=14 %s' % params)
        system('conan create demo/testing -s compiler="Visual Studio" -s compiler.
↪version=12 %s' % params)
        system('conan create demo/testing -s compiler="gcc" -s compiler.version=4.8 %s
↪' % params)
    else:
        pass
```

This is a pure python script, not related to conan, and should be run as such:

```
$ python build.py
```

We have developed another FOSS tool for package creators, **Conan Package Tools** to ease the task of generating multiple binary packages from a package recipe. It offers a simple way to define the different configurations and to call **`conan test`**. Also offers CI integration like **Travis CI, Appveyor and Bamboo**, for cloud based automated binary package creation, testing and uploading.

This tool enables the creation of hundreds of binary packages in the cloud with a simple `$ git push`.

- Make easier the **generation of multiple conan packages** with different configurations.

- Automated/remote package generation in **Travis/Appveyor** server with distributed builds in CI jobs for big/slow builds.
- **Docker**: Automatic generation of packages for several versions of `gcc` and `clang` in Linux, also in Travis CI.
- Automatic creation of OSX packages with apple-clang, also in Travis-CI.
- **Visual Studio**: Automatic configuration of command line environment with detected settings.

It's available in pypi:

```
$ pip install conan_package_tools
```

Read the README.md in the Conan Package Tools repository.

CHAPTER

SEVEN

UPLOADING PACKAGES

This section shows how to upload packages using remotes as well as the different binary repositories you can use.

7.1 Remotes

In the previous sections, we built several packages in our computer, those packages are stored in the local cache, typically *~/.conan/data*. Now, you might want to upload them to a conan server for later reuse on another machine, project, or for sharing them.

Conan packages can be uploaded to different remotes previously configured with a name and an URL. The remotes are just servers used as binary repositories that store packages by reference.

There are several possibilities to have a server where to upload packages:

For private development:

- **Artifactory Community Edition for C/C++**: Artifactory Community Edition (CE) for C/C++ is a completely free Artifactory server that implements both Conan and generic repositories. It is the recommended server for companies and teams wanting to host their own private repository. It has a web UI, advanced authentication and permissions, very good performance and scalability, a Rest API, it can host generic artifacts (tarballs, zips, etc). Check *Artifactory Community Edition for C/C++* for more information.
- **Artifactory Pro**: Artifactory is the binary repository manager for all major packaging formats. It is the recommended remote type for enterprise and professional package management. Check Artifactory documentation for more information. For comparison between Artifactory editions, check the Artifactory Comparison Matrix.
- **Conan server**: Simple, free and open source, MIT licensed server that comes bundled with the conan client. Check *Running conan_server* for more information.

For distribution:

- **Bintray**: Bintray is a cloud platform that gives you full control over how you publish, store, promote, and distribute software. You can create binary repositories in Bintray to share conan packages or even create an organization. It is free for open source packages, and the recommended server to distribute them to the C and C++ communities. Check *Using Bintray* for more information.

7.1.1 Bintray official repositories

Conan official repositories for open source libraries are hosted in Bintray. These repositories are maintained by the Conan team. Currently there are two central repositories:

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

This repository has moderated, curated and well-maintained packages, and is the place where you can share your packages with the community. To share your package, you can upload it to your own (or your organization's) repositories and submit an inclusion request to conan-center. Check *conan-center guide* for more information.

conan-transit: https://bintray.com/conan/conan-transit (DEPRECATED)

Deprecated. Contains mostly outdated packages some of them even not compatible with latest Conan versions, so its usage is discouraged. This repository only exists for backwards compatibility, it is not a default remote in the Conan client and will be completely removed soon. This repository is an exact copy of the old `server.conan.io` repository at **June 11, 2017 08:00 CET**. It's a read-only repository, so you can only download hosted packages.

Conan comes with **conan-center** repository configured by default. Just in case you want to manually configure this repository you can always add it like this:

```
$ conan remote add conan-center https://conan.bintray.com
```

7.1.2 Bintray community repositories

There are some popular community repositories that may be of interest for conan users to retrieve open source packages from. Some of these repositories are not affiliated with the Conan team.

bincrafters : https://bintray.com/bincrafters/public-conan

The Bincrafters team builds binary software packages for the OSS community. This repository contains a wide and growing variety of conan packages from contributors.

Use the following command to add this remote to Conan:

```
$ conan remote add bincrafters https://api.bintray.com/conan/bincrafters/
↪public-conan
```

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

Created by Conan developers, it should be considered as an incubator to mature packages before contacting authors or including them in conan-center. This repository contains work-in-progress packages that may still not work and may not be fully featured.

Use the following command to add this remote to Conan:

```
$ conan remote add conan-community https://api.bintray.com/conan/conan-
↪community/conan
```

Note: If you are working in a team, you probably want to use the same remotes everywhere: developer machines, CI. The `conan config install` command can automatically define the remotes in a conan client, as well as other resources as profiles. Have a look to the *conan config install* command.

7.2 Uploading packages to remotes

First, check if the remote you want to upload to is already in your current remote list:

```
$ conan remote list
```

You can add any remote easily. For a remote running in your machine, you could run:

```
$ conan remote add my_local_server http://localhost:9300
```

You can search any remote in the same way you search your computer. Actually, many conan commands can specify a specific remote.

```
$ conan search -r=my_local_server
```

Now, upload the package recipe and all the packages to your remote. In this example we are using our `my_local_server` remote, but you could use any other.

```
$ conan upload Hello/0.1@demo/testing --all -r=my_local_server
```

You might be prompted for a username and password. The default conan server remote has a **demo/demo** account we can use for testing.

The `--all` option will upload the package recipe plus all the binary packages. Now try again to read the information from the remote (we refer to it as remote, even if it is running on your local machine, as it could be run on another server in your LAN):

```
$ conan search Hello/0.1@demo/testing -r=my_local_server
```

Note: If package upload fails, you can try to upload it again. Conan keeps track of the upload integrity and will only upload missing files.

Now we can check if we are able to download and use them in a project. For that purpose, we first have to **remove the local copies**, otherwise the remote packages will not be downloaded. Since we have just uploaded them, they are identical to the local ones.

```
$ conan remove Hello*
$ conan search
```

Since we have our test setup from the previous section, we can just use it for our test. Go to your package folder and run the tests again, now saying that we don't want to build the sources again, we just want to check if we can download the binaries and use them:

```
$ conan create . demo/testing --not-export --build=never
```

You will see that the test is built, but the packages are not. The binaries are simply downloaded from your local server. You can check their existence on your local computer again with:

```
$ conan search
```

7.3 Using Bintray

On Bintray, you can create and manage as many free, personal Conan repositories as you like. On an OSS account, all packages you upload are public, and anyone can use them by simply adding your repository to their Conan remotes.

To allow collaboration on open source projects, you can also create Organizations in Bintray and add members who will be able to create and edit packages in your organization's repositories.

7.3.1 Uploading to Bintray

Conan packages can be uploaded to Bintray under your own users or organizations. To create a repository you can follow these steps:

1. **Create a Bintray Open Source account**

   Browse to https://bintray.com/signup/oss and submit the form to create your account. Note that you don't have to use the same username that you had in your Conan account.

   > **Warning:** Please **make sure you use the Open Source Software OSS account**. Follow this link: https://bintray.com/signup/oss. Bintray provides free conan repositories for OSS projects, no need to open a Pro or Enterprise Trial account.

2. **Create a Conan repository**

   If you intend to collaborate with other users, you first need to create a Bintray organization, and create your repository under the organization's profile rather than under your own user profile.

   On your user profile (or organization profile) click "Add new repository". Fill in the Create Repository form making sure to select Conan as the Type.

3. **Add your Bintray repository**

   Add a Conan remote in your Conan client pointing to your Bintray repository

   ```
   $ conan remote add <REMOTE> <YOUR_BINTRAY_REPO_URL>
   ```

   Use the Set Me Up button on your repository's page on Bintray to get its URL.

4. **Get your API key**

   Your API key is the "password" used to authenticate the Conan client to Bintray, NOT your Bintray password. To get your API key, you need to go to "Edit Your Profile" in your Bintray account and check the API Key section.

5. **Set your user credentials**

   Add your conan user with the API Key, your remote and your Bintray user name:

   ```
   $ conan user -p <APIKEY> -r <REMOTE> <USERNAME>
   ```

Setting the remotes in this way will make your Conan client resolve packages and install them from repositories in the following order of priority:

1. conan-center
2. Your own repository

If you want to have your own repository prioritized, please use the `--insert` command line option when adding it:

```
$ conan remote add <your_remote> <your_url> --insert 0
$ conan remote list
  <your remote>: <your_url> [Verify SSL: True]
  conan-center: https://conan.bintray.com [Verify SSL: True]
```

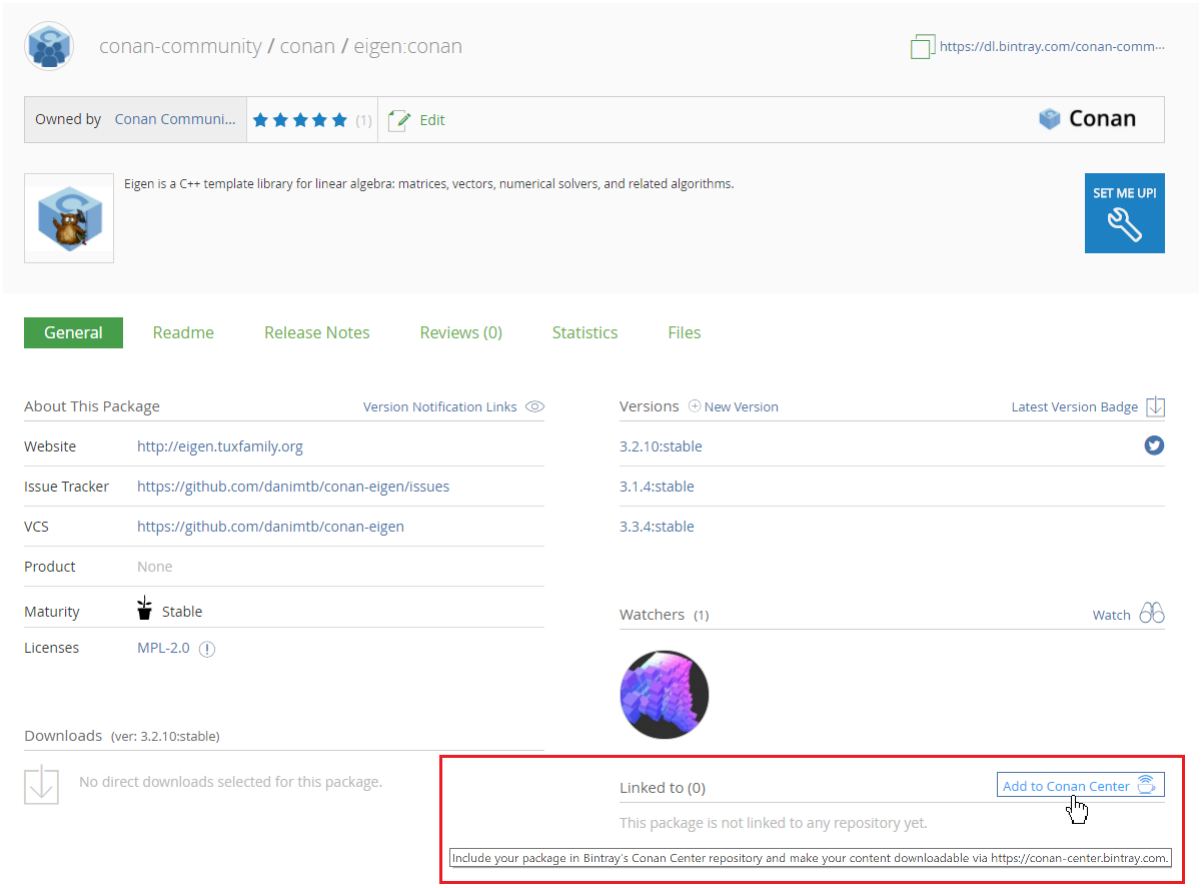
Tip: Check the full reference of *$ conan remote* command.

7.3.2 Contributing packages to conan-center

As a moderated and curated repository, conan-center will not be populated automatically. Initially, it will be empty. To have your recipe or binary packages available on conan-center, you need to submit an inclusion request to Bintray, and the Bintray team will review your request.

- If you are the **author of an open source library**, your package will be approved. Keep in mind that it is your responsibility to maintain acceptable standards of quality for all packages you submit for inclusion in conan-center.
- If you are packaging a third-party library you need to follow the guidelines below.

When you know how to *upload your packages to your own Bintray repository*, contributing a library to Conan Center is really straightforward. All you have to do is to navigate to the main page of the package in Bintray and click the **"Add to Conan Center"** button to start the inclusion request process.



Inclusion guidelines for third party libraries

In the inclusion request process, the JFrog staff will perform a general review and will make suggestions for improvements or better/cleaner ways to implement the package.

One conan package per OSS library

Before creating packages for third party libraries, please read these general guidelines.

- Ensure that there is no other conan package for the same library. If you are planning to support a new version of a library that already exists in the `conan-center` repository, please, contact the package author and collaborate. All the versions of the same library have to be on the same Bintray Conan package.
- It is recommended to contact the **library author** and suggest to maintain the Conan package. When possible, open a pull request to the original repository of the library with the conan needed files or suggest to open a new repository with the recipe.
- If you are going to collaborate with different users to maintain the Conan package, open a Bintray organization.

Recipe quality

- **Git public repository**: The recipe needs to be hosted in a public Git repository that allows collaboration.
- **Recipe fields**: *description*, *license* and *url* are required. The *license* field refers to the library being packaged.
- **Linter:** Is important to have a reasonable clean Linter, **`conan export`** and **`conan create`** will output some warnings and errors, keep it as clean as possible to guarantee a recipe less error prone and more understandable.
- **Updated:** Not using deprecated features and when possible, using latest conan features, build helpers etc.
- **Clean:** The code style will be reviewed to guarantee the readability of the recipe.
- **test_package:** The recipes must contain a *test_package*
- **Maintenance commitment:** You will be the responsible to keep the recipe updated, fix issues etc., so a minimal commitment will be required. Conan organization reserves the right to unlink a poorly maintained package or replace it with better alternatives.
- **Raise errors on invalid configurations:** If the library doesn't work for a specific configuration, e.g. requires **gcc>7** the recipe must contain a `configure(self)` method that raises an exception in case of invalid settings/options.

```
def configure():
    if self.settings.compiler == "gcc" and self.settings.compiler.version < "7.0":
        raise ConanException("GCC > 7.0 is required")
    if self.settings.os == "Windows":
        raise ConanException("Windows not supported")
```

- **Without version ranges**: As many libraries does not follow semantic versioning and the dependency resolution of version ranges is not always clear, recipes in conan center should fix the version of their dependencies and not use version ranges.
- **LICENSE of the recipe:** The public repository must contain a `LICENSE` file with an OSS license.
- **LICENSE of the library:** Every built binary package must contain one or more `license*` file(s), so make sure that in the `package()` method of your recipe, you are copying the library licenses to a `licenses` subfolder.

```
def package():
    self.copy("license*", dst="licenses",  ignore_case=True, keep_path=False)
```

Sometimes there is no `license` file, and you need to extract the license from a header file, this is an example:

```
def package():
    # Extract the License/s from the header to a file
    tmp = tools.load("header.h")
    license_contents = tmp[2:tmp.find("*/", 1)] # The license begins with a C comment␣
↪/* and ends with */
```

(continues on next page)

(continued from previous page)

```
    tools.save("LICENSE", license_contents)

    # Package it
    self.copy("license*", dst="licenses",  ignore_case=True, keep_path=False)
```

CI Integration

- If you are packaging a header only library, it is only needed to provide one CI configuration (e.g. Travis with gcc 6.1) to check that the package is built correctly (use **`conan create`**).
- Unless your library is a header only library or doesn't support a concrete operating system or compiler you will need to provide a CI systems integration to support:
  - **Linux:** GCC, desirable latest version from each major (4.9, 5.4, 6.3)
  - **Linux:** Clang, desirable latest version from each major (3.9, 4.0)
  - **Mac OSX:** Two latest versions of apple-clang, e.j (8.0, 8.1) or newer.
  - **Windows:** Visual Studio 12, 14 and 15 (or newer)
- The easiest way to provide the CI integration (with Appveyor for Windows builds, Travis.ci for Linux and OSX, and Gitlab for Linux) is to use the *conan new* command. Take a look to the options to generate a library layout with the needed appveyor/travis/gitlab.

  You can also copy the following files from this zlib Conan package repository and adapt them:
  - `.travis` folder. Not needed to adjust anything.
  - `.travis.yml` file. Adjust your username, library reference etc
  - `appveyor.yml` file. Adjust your username, library reference etc
- Take a look to the *Travis CI*, *Appveyor* and *GitLab CI* integration guides.

Bintray package information

In the bintray page of your package fill the following fields:

- Description (description of the packaged library)
- Licenses (license of the packaged library)
- Tags
- Maturity
- Website: If any, website of the library
- Issues tracker: URL of the issue tracker from your github repository e.j: https://github.com/conan-community/conan-zlib/issues
- Version control: URL of your recipe github repository. e.j: https://github.com/conan-community/conan-zlib
- GitHub repo (user/repo): e.j lasote/conan-zlib

In each version page (optional, but welcomed):

- Select the README from github.
- Select the Release Notes.

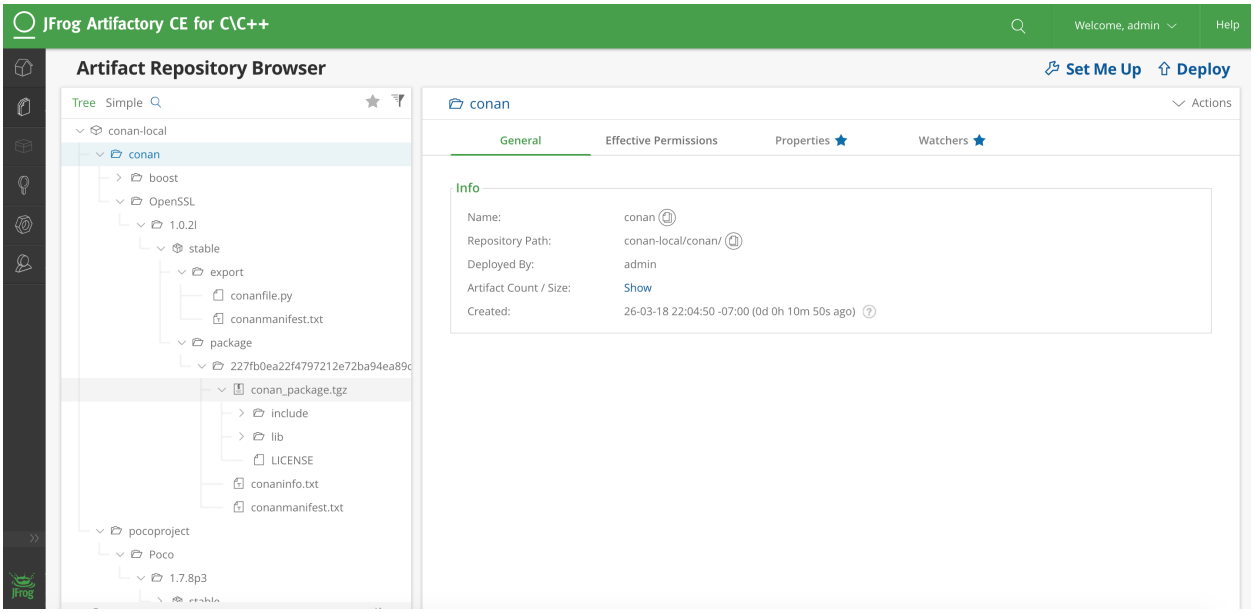
7.4 Artifactory Community Edition for C/C++

Artifactory Community Edition (CE) for C/C++ is the recommended server for development and hosting private packages for a team or company. It is completely free, and it features a WebUI, advanded authentication and permissions, great performance and scalability, a Rest API, a generic CLI tool and generic repositories to host any kind of source or binary artifact.

This is a very brief introduction to Artifactory CE, for the complete Artifactory CE documentation, visit Artifactory docs.

7.4.1 Running Artifactory CE

There are several ways to download and run Artifactory CE. The simplest one might be to download and unzip the given zip file, though other installers, included running from a docker image, are available. When the file is unzipped, Artifactory can be launched double clicking on the .bat or .sh script in the *bin* subfolder, depending on the OS. Java 8 update 45 or later runtime is required, if you don't have it, please install it first (newer Java versions preferred).



Once Artifactory has started, navigate to the default URL *http://localhost:8081*, where the Web UI should be running. The default user and password are `admin:password`.

7.4.2 Creating and using a conan repo

Navigate to Admin -> Repositories -> Local, then click on the "New" button. A dialog for selecting the package type will appear, select Conan, then type a "Repository Key" (the name of the repository you are about to create), for example "conan-local". You can create multiple repositories to serve different flows, teams, or projects.

Now, it is necessary to set-up the client. Go to Artifacts, and click on the created repository. The "Set Me Up" button in the top right corner will give instructions how to configure the remote in the conan client:

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

From now, you can upload, download, search, etc. this remote as any other one.

```
$ conan upload * --all -r=artifactory
$ conan search * -r=artifactory
```

7.4.3 Migrating from other servers

If you are already running another server, like the open source *conan_server*, it is very easy to migrate your packages, using the conan client to download the packages and re-upload them to the new server.

This python script might be helpful, given that it had already defined the respective `local` and `artifactory` remotes:

```
import os
import subprocess

def run(cmd):
ret = os.system(cmd)
if ret != 0:
    raise Exception("Command failed: %s" % cmd)

# Assuming local = conan_server and artifactory remotes
output = subprocess.check_output("conan search -r=local --raw")
packages = output.splitlines()

for package in packages:
print("Downloading %s" % package)
run("conan download %s -r=local" % package)

run("conan upload * --all --confirm -r=artifactory")
```

7.5 Running conan_server

conan_server is a free and open source server that implements conan remote repositories. It is a very simple application, bundled with the regular conan client installation. For most cases, it is recommended to use the free Artifactory Community Edition for C/C++ server, check *Artifactory Community Edition for C/C++* for more information.

Running the simple open source *conan_server* that comes with the conan installers (or pip packages) is simple. Just open a terminal and type:

```
$ conan_server
```

Note: On Windows, you might experience problems with the server, if you run it under bash/msys. It is better to launch it in a regular `cmd` window.

This server is mainly for testing (though it might work fine for small teams). If you need a more stable, responsive and robust server, you should run it from source:

7.5.1 Running from source (linux)

The conan installer includes a simple executable **conan_server** for a server quick start. But you can use the **conan server** through the WSGI application, which means that you can use gunicorn to run the app, for example.

First, clone the conan repository from source and install the requirements:

```
$ git clone https://github.com/conan-io/conan.git
$ cd conan
$ git checkout master
$ pip install -r conans/requirements.txt
$ pip install -r conans/requirements_server.txt
$ pip install gunicorn
```

Run the server application with `gunicorn`. In the following example we will run server on port 9300 with 4 workers and a timeout of 5 minutes (300 seconds, for large uploads/downloads, you can also decrease it if you don't have very large binaries):

```
$ gunicorn -b 0.0.0.0:9300 -w 4 -t 300 conans.server.server_launcher:app
```

Note: Please note the timeout of `-t 300` seconds, 5 minutes parameter. If your transfers are very large or in a slow network, you might need to increase that value.

You can also bind to an IPV6 address or specify both IPv4 and IPv6 addresses:

```
$ gunicorn -b 0.0.0.0:9300 -b [::1]:9300 -w 4 -t 300 conans.server.server_launcher:app
```

7.5.2 Server configuration

Your server configuration lives in `~/.conan_server/server.conf`. You can change values there, prior to launching the server. Note that the server is not reloaded when the values are changed. You have to stop and restart it manually.

The server configuration file is by default:

```
[server]
jwt_secret: MnpuzsExftskYGOMgaTYDKfw
jwt_expire_minutes: 120

ssl_enabled: False
port: 9300
public_port:
host_name: localhost

store_adapter: disk
authorize_timeout: 1800

# Just for disk storage adapter
disk_storage_path: ~/.conan_server/data
disk_authorize_timeout: 1800

updown_secret: NyiSWNWnwumTVpGpoANuyyhR


[write_permissions]
# "opencv/2.3.4@lasote/testing": default_user,default_user2

[read_permissions]
# opencv/1.2.3@lasote/testing: default_user default_user2
```

(continues on next page)

(continued from previous page)

```
# By default all users can read all blocks
*/*@*/*: *

[users]
demo: demo
```

Server parameters

Note: Conan server from v1.1 supports relative URLs, so you can avoid setting `host_name`, `public_port` and `ssl_enabled`. The URLs used to upload/download packages will be automatically generated in the client following the URL of the remote. It allows to access the conan server from different networks.

- `port`: Port where **conan_server** will run.
- The client server authorization is done with JWT. `jwt_secret` is a random string used to generate authentication tokens. You can change it safely anytime (in fact it is a good practice), the change will just force users to log in again. `jwt_expire_minutes` is the amount of time that users remain logged-in within the client without having to introduce their credentials again.

Other parameters (not recommended from Conan 1.1, but necessary for previous versions):

- `host_name` If you set `host_name` you must use the machine's IP where you are running your server (or domain name), something like **host_name: 192.168.1.100**. This IP (or domain name) has to be visible (and resolved) by the conan client, so take it in account if your server has multiple network interfaces.
- `public_port` Which might be needed if running virtualized, docker or any other kind of port redirection. Files uploads/downloads are served with their own URLs, generated by the system, so the file storage backend is independent. Those URLs need the public port they have to communicate from the outside. If you leave it blank, it will use the `port` value.

  Example: Use conan_server in a docker container that internally runs in the 9300 port but it exposes the 9999 port (where the clients will connect to):

  ```
  docker run ... -p9300:9999 ... # Check Docker docs for that
  ```

  server.conf

  ```
  [server]

  ssl_enabled: False
  port: 9300
  public_port: 9999
  host_name: localhost
  ```

- `ssl_enabled` Conan doesn't handle the SSL traffic by itself, but you can use a proxy like nginx to redirect the SSL traffic to your conan server. If your conan clients are connecting with "https" set *ssl_enabled* to True. This way conan_server will generate the upload/download urls with "https" instead of "http".

Note: **Important**: Conan client, by default, will validate the server SSL certificates and won't connect if it's not valid. If you have self signed certificates you have two options:

1. Use the **`conan remote`** command to disable the SSL certifate checks. e.j: *conan remote add/update myremote https://somedir False*

2. Append the server *.crt* file contents to *~/.conan/cacert.pem* file.

Check the section *How to manage SSL (TLS) certificates* section to know more about it.

Conan has implemented an extensible storage backend, based on the abstract class `StorageAdapter`. Currently the server only supports storage in `disk`. The folder in which uploaded packages are stored (i.e., the folder you would want to backup) is defined in `disk_storage_path`.

The storage backend might use a different channel, and uploads/downloads are authorized up to a maximum of `authorize_timeout` seconds. The value should be enough so large downloads/uploads are not rejected, but not too big to prevent hanging up the file transfers. The value `disk_authorize_timeout` is not currently used. File transfers are authorized with their own tokens, generated with the secret `updown_secret`. This value should be different from the above `jwt_secret`.

Running conan server with SSL using nginx

server.conf

```
[server]
port: 9300
```

nginx conf file

```
server {
    listen 443;
    server_name myservername.mydomain.com;

    location / {
      proxy_pass http://0.0.0.0:9300;
    }
    ssl on;
    ssl_certificate /etc/nginx/ssl/server.crt;
    ssl_certificate_key /etc/nginx/ssl/server.key;
}
```

remote configuration in Conan client

```
$ conan remote add myremote https://myservername.mydomain.com/subdir
```

Running conan server with SSL using nginx in a subdirectory

server.conf

```
[server]
port: 9300
```

nginx conf file

```
server {

       listen 443;
       ssl on;
       ssl_certificate /usr/local/etc/nginx/ssl/server.crt;
       ssl_certificate_key /usr/local/etc/nginx/ssl/server.key;
       server_name myservername.mydomain.com;
```

(continues on next page)

(continued from previous page)

```
      location ~/subdir/(.*)$ {
         proxy_pass http://0.0.0.0:9300/$1;
      }
  }
```

remote configuration in Conan client

```
$ conan remote add myremote https://myservername.mydomain.com/subdir
```

Running conan server using Apache

You need to install `mod_wsgi`. If you want to use Conan installed from `pip`, the conf file should be roughly as follows:

Apache conf file (e.j /etc/apache2/sites-available/0_conan.conf)

```
<VirtualHost *:80>
    WSGIScriptAlias / /usr/local/lib/python2.7/dist-packages/conans/server/
↪server_launcher.py
    WSGICallableObject app
    WSGIPassAuthorization On

    <Directory /usr/local/lib/python2.7/dist-packages/conans>
        Require all granted
    </Directory>
</VirtualHost>
```

If you want to use Conan checked out from source in, say, */srv/conan*, the conf file should be as follows:

Apache conf file (e.j /etc/apache2/sites-available/0_conan.conf)

```
<VirtualHost *:80>
    WSGIScriptAlias / /srv/conan/conans/server/server_launcher.py
    WSGICallableObject app
    WSGIPassAuthorization On

    <Directory /srv/conan/conans>
        Require all granted
    </Directory>
</VirtualHost>
```

The directive `WSGIPassAuthorization On` is needed to pass the HTTP basic authentication to Conan.

Also take into account that the server config files are located in the home of the configured Apache user, e.j var/www/.conan_server, so remember to use that directory to configure your conan server.

Permissions parameters

By default, the server configuration is: Read can be done anonymous, but uploading requires registered users. Users can be easily registered in the `[users]` section, defining a pair of `login: password` for each one. Yes, plain text passwords at the moment, but as the server is on-premises (behind firewall), you just need to trust your sysadmin :)

If you want to restrict read/write access to specific packages, configure it in the `[read_permissions]` and `[write_permissions]` sections. These sections allow a sequence of patterns and allowed users, in the form:

```
# use a comma-separated, no-spaces list of users
package/version@user/channel: allowed_user1,allowed_user2
```

E.g.:

```
*/*@*/*: * # allow all users to all packages
PackageA/*@*/*: john,peter # allow john and peter access to any PackageA
*/*@project/*: john # Allow john to access any package from the "project" user
```

The rules are evaluated in order, if the left side of the pattern matches, the rule is applied and it will not look further.

Authentication

Conan provides by default a simple `user: password` users list in the `server.conf` file.

There is also a plugin mechanism for setting other authentication methods. The process to install any of them is a simple 2 step process:

1. Copy the authenticator source file into the `.conan_server/plugins/authenticator` folder
2. Add `custom_authenticator: authenticator_name` in the `server.conf` [server] section

This is a list of available authenticators, visit their URLs to get them, but also to report issues and collaborate:

- **htpasswd**: Use your server Apache htpasswd file to authenticate users. Get it: https://github.com/d-schiffner/conan-htpasswd
- **LDAP**: Use your LDAP server to authenticate users. Get it: https://github.com/uilianries/conan-ldap-authentication

Create your own custom Authenticator

If you want to create your own Authenticator, create a python module in `~/.conan_server/plugins/authenticator/my_authenticator.py`

Example:

```
def get_class():
    return MyAuthenticator()


class MyAuthenticator(object):
    def valid_user(self, username, plain_password):
        return username == "foo" and plain_password == "bar"
```

The module has to implement:

- A factory function `get_class()` that returns a class with a `valid_user()` method instance.
- The class containing the `valid_user()` that has to return True if the user and password are valid or False otherwise.

Got any doubts? Please check out our *FAQ section* or .

CHAPTER

EIGHT

PACKAGE APPS AND DEVTOOLS

With conan it is possible to package and deploy applications. It is also possible that these applications are also dev-tools, like compilers (e.g. MinGW), or build systems (e.g. CMake).

This section describes how to package and run executables, and also how to package dev-tools. Also, how to apply applications like dev-tools or even libraries (like testing frameworks) to other packages to build them from sources:`build_requires`

8.1 Running and deploying packages

Executables and applications including shared libraries can be also distributed, deployed and run with conan. This might have some advantages compared to deploying with other systems:

- A unified development and distribution tool, for all systems and platforms
- Manage any number of different deployment configurations in the same way you manage them for development
- Use a conan server remote to store all your applications and runtimes for all Operating Systems, platforms and targets

There are different approaches:

8.1.1 Using virtual environments

We can crate a package that contains an executable, for example from the default package template created by **`conan new`**:

```
$ conan new Hello/0.1
```

The source code used contains an executable called `greet`, but it is not packaged by default. Let's modify the recipe `package()` method to also package the executable:

```
def package(self):
    self.copy("*greet*", src="hello/bin", dst="bin", keep_path=False)
```

Now, we can create the package as usual, but if we try to run the executable, it won't be found:

```
$ conan create . user/testing
...
Hello/0.1@user/testing package(): Copied 1 '.h' files: hello.h
Hello/0.1@user/testing package(): Copied 1 '.exe' files: greet.exe
Hello/0.1@user/testing package(): Copied 1 '.lib' files: hello.lib
```

(continues on next page)

(continued from previous page)

```
$ greet
> ... not found...
```

Conan does not modify by default the environment, it will just create the package in the local cache, and that is not in the system PATH, so the `greet` executable is not found.

The `virtualrunenv` generator will generate files that add the packages default binary locations to the necessary paths:

- It adds the dependencies `lib` subfolder to the `DYLD_LIBRARY_PATH` environment variable (for OSX shared libraries)
- It adds the dependencies `lib` subfolder to the `LD_LIBRARY_PATH` environment variable (for Linux shared libraries)
- It adds the dependencies `bin` subfolder to the `PATH` environment variable (for executables)

So if we install the package, specifying such `virtualrunenv` like:

```
$ conan install Hello/0.1@user/testing -g virtualrunenv
```

We will get some files, that can be called to activate and deactivate such environment variables

```
$ activate_run.sh # $ source activate_run.sh in Unix/Linux
$ greet
> Hello World!
$ deactivate_run.sh # $ source deactivate_run.sh in Unix/Linux
```

8.1.2 Imports

It is possible to define a custom conanfile (either .txt or .py), with an `imports` section, that can retrieve from local cache the desired files. This approach, requires an user conanfile. For more details see example below *runtime packages*

8.1.3 Deployable packages

With the `deploy()` method, a package can specify which files and artifacts to copy to user space or to other locations in the system. Let's modify the example recipe adding the `deploy()` method:

```
def deploy(self):
    self.copy("*", dst="bin", src="bin")
```

And run **`conan create`**

```
$ conan create . user/testing
```

With that method in our package recipe, it will copy the executable when installed directly:

```
$ conan install Hello/0.1@user/testing
...
> Hello/0.1@user/testing deploy(): Copied 1 '.exe' files: greet.exe
$ bin\greet.exe
> Hello World!
```

The deploy will create a `deploy_manifest.txt` file with the files that have been deployed.

Sometimes it is useful to adjust the package ID of the deployable package in order to deploy it regardless of the compiler it was compiled with:

```
def package_id(self):
    del self.info.settings.compiler
```

See also:

Read more about the *deploy()* method.

8.1.4 Running from packages

If you want to directly run one executable from your dependencies, it is not necessary to use the generators and activate the environment, as it can be directly done in code with the `RunEnvironment` helper. So if the `Consumer` package is willing to execute the `greet` app while building its own package, it can be done:

```
from conans import ConanFile, tools, RunEnvironment

class ConsumerConan(ConanFile):
    name = "Consumer"
    version = "0.1"
    settings = "os", "compiler", "build_type", "arch"
    requires = "Hello/0.1@user/testing"

    def build(self):
        env = RunEnvironment(self)
        with tools.environment_append(env.vars):
            self.run("greet")
```

Now run **conan install** and **conan build** for this consumer recipe:

```
$ conan install . && conan build .
...
Project: Running build()
Hello World!
```

Instead of using the environment, it is also possible to access the path of the dependencies:

```
def build(self):
    path = os.path.join(self.deps_cpp_info["Hello"].rootpath, "bin")
    self.run("%s/greet" % path)
```

Note, however, that this might not be enough if shared libraries exist, while using the above `RunEnvironment` is a more complete solution

Finally, there is another approach: the package containing the executable, adds its "bin" folder to the PATH. In this case the **Hello** package conanfile would contain:

```
def package_info(self):
    self.cpp_info.libs = ["hello"]
    self.env_info.PATH = os.path.join(self.package_folder, "bin")
```

Note that this is not enough for shared libraries, and defining DYLD_LIBRARY_PATH and LD_LIBRARY_PATH could be necessary.

The consumer package would be simple, as the PATH environment variable will already contain the desired path to greet executable:

```
def build(self):
    self.run("greet")
```

8.1.5 Runtime packages and re-packaging

It is possible to create packages that contain only runtime binaries, getting rid of all build-time dependencies. If we want to create a package from the above "Hello" one, but only containing the executable (rembember that the above package also contains a library, and the headers), we could do:

```
from conans import ConanFile

class HellorunConan(ConanFile):
    name = "HelloRun"
    version = "0.1"
    build_requires = "Hello/0.1@user/testing"
    keep_imports = True

    def imports(self):
        self.copy("*.exe", dst="bin")

    def package(self):
        self.copy("*")
```

This recipe has the following characteristics:

- It includes the `Hello/0.1@user/testing` package as `build_requires`. That means that it will be used to build this "HelloRun" package, but once the "HelloRun" package is built, it will not be necessary to retrieve it.
- It is using an `imports()` to copy from the dependencies, in this case, the executable
- It is using `keep_imports` attribute to define that imported artifacts during the `build()` step (which is not define, then using the default empty one), are kept and not removed after build
- The `package()` method packages the imported artifacts that will be in the build folder.

To create and upload to remote this package:

```
$ conan create . user/testing
$ conan upload HelloRun* --all -r=my-remote
```

After that, installing and running this package, can be done by any of the means presented above, for example, we could do:

```
$ conan install HelloRun/0.1@user/testing -g virtualrunenv
# You can specify the remote with -r=my-remote
# It will not install Hello/0.1@...
$ activate_run.sh # $ source activate_run.sh in Unix/Linux
$ greet
> Hello World!
$ deactivate_run.sh # $ source deactivate_run.sh in Unix/Linux
```

8.2 Creating conan packages to install dev tools

Conan 1.0 introduced two new settings, `os_build` and `arch_build`. These settings represent the machine where Conan is running, and are important settings when we are packaging tools.

These settings are different from `os` and `arch`. These mean where the built software by the Conan recipe will run. When we are packaging a tool, it usually makes no sense, because we are not building any software, but it makes sense if you are *cross building software*.

We recommend the use of `os_build` and `arch_build` settings instead of `os` and `arch` if you are packaging a tool involved in the building process, like a compiler, a build system etc. If you are building a package to be run on the **host** system you can use `os` and `arch`.

A Conan package for a tool follows always a similar structure, this is a recipe for packaging the `nasm` tool for building assembler:

```
import os
from conans import ConanFile
from conans.client import tools


class NasmConan(ConanFile):
    name = "nasm"
    version = "2.13.01"
    license = "BSD-2-Clause"
    url = "https://github.com/lasote/conan-nasm-installer"
    settings = "os_build", "arch_build"
    build_policy = "missing"
    description="Nasm for windows. Useful as a build_require."

    def configure(self):
        if self.settings.os_build != "Windows":
            raise Exception("Only windows supported for nasm")

    @property
    def nasm_folder_name(self):
        return "nasm-%s" % self.version

    def build(self):
        suffix = "win32" if self.settings.arch_build == "x86" else "win64"
        nasm_zip_name = "%s-%s.zip" % (self.nasm_folder_name, suffix)
        tools.download("http://www.nasm.us/pub/nasm/releasebuilds/"
                       "%s/%s/%s" % (self.version, suffix, nasm_zip_name), nasm_zip_
↪name)
        self.output.warn("Downloading nasm: "
                         "http://www.nasm.us/pub/nasm/releasebuilds"
                         "/%s/%s/%s" % (self.version, suffix, nasm_zip_name))
        tools.unzip(nasm_zip_name)
        os.unlink(nasm_zip_name)

    def package(self):
        self.copy("*", dst="", keep_path=True)
        self.copy("license*", dst="", src=self.nasm_folder_name, keep_path=False,␣
↪ignore_case=True)

    def package_info(self):
        self.output.info("Using %s version" % self.nasm_folder_name)
        self.env_info.path.append(os.path.join(self.package_folder, self.nasm_folder_
↪name))
```

(continues on next page)

(continued from previous page)

There are some remarkable things in the recipe:

- The configure method discards some combinations of settings and options, by throwing an exception. In this case this package is only for Windows.
- `build()` downloads the appropriate file and unzips it.
- `package()` copies all the files from the zip to the package folder.
- `package_info()` uses `self.env_info` to append to the environment variable `path` the package's bin folder.

This package has only 2 differences from a regular Conan library package:

- `source()` method is missing. That's because when you compile a library, the source code is always the same for all the generated packages, but in this case we are downloading the binaries, so we do it in the build method to download the appropriate zip file according to each combination of settings/options. Instead of actually building the tools, we just download them. Of course, if you want to build it from source, you can do it too by creating your own package recipe.
- The `package_info()` method uses the new `self.env_info` object. With `self.env_info` the package can declare environment variables that will be set automatically before *build()*, *package()*, *source()* and *imports()* methods of a package requiring this build tool. This is a convenient method to use these tools without having to mess with the system path.

8.2.1 Using the tool packages in other recipes

The `self.env_info` variables will be automatically applied when you require a recipe that declares them. For example, take a look at the MinGW *conanfile.py* recipe (https://github.com/conan-community/conan-mingw-installer):

```
class MingwInstallerConan(ConanFile):
    name = "mingw_installer"
    ...

    build_requires = "7z_installer/1.0@conan/stable"

    def build(self):
        keychain = "%s_%s_%s_%s" % (str(self.settings.compiler.version).replace(".",
↪""),
                                    self.settings.arch_build,
                                    self.settings.compiler.exception,
                                    self.settings.compiler.threads)

        files = {
           ...        }

        tools.download(files[keychain], "file.7z")
        self.run("7z x file.7z")

    ...
```

We are requiring a `build_require` to another package: `7z_installer`. In this case it will be used to unzip the 7z compressed files after downloading the appropriate MinGW installer.

That way, after the download of the installer, the 7z executable will be in the PATH, because the `7z_installer` dependency declares the *bin* folder in its `package_info()`.

Important: Some build requires will need settings such as `os`, `compiler` or `arch` to build themselves from sources. In that case the recipe might look like this:

```
class MyAwesomeBuildTool(ConanFile):
    settings = "os_build", "arch_build", "arch", "compiler"
    ...

    def build(self):
        cmake = CMake(self)
        ...

    def package_id(self):
        self.info.include_build_settings()
        del self.info.settings.compiler
        del self.info.settings.arch
```

Note `package_id()` deletes not needed information for the computation of the package ID and includes the build settings `os_build` and `arch_build` that are excluded by default. Read more about *self.info.include_build_settings()* in the reference section.

8.2.2 Using the tool packages in your system

You can use the *virtualenv generator* to get the requirements applied in your system. For example: Working in Windows with MinGW and CMake.

1. Create a separate folder from your project, this folder will handle our global development environment.

```
$ mkdir my_cpp_environ
$ cd my_cpp_environ
```

2. Create a *conanfile.txt* file:

```
[requires]
mingw_installer/1.0@conan/stable
cmake_installer/3.10.0@conan/stable

[generators]
virtualenv
```

Note that you can adjust the `options` and retrieve a different configuration of the required packages, or leave them unspecified in the file and pass them as command line parameters.

3. Install them:

```
$ conan install .
```

4. Activate the virtual environment in your shell:

```
$ activate
(my_cpp_environ)$
```

5. Check that the tools are in the path:

```
(my_cpp_environ)$ gcc --version

> gcc (x86_64-posix-seh-rev1, Built by MinGW-W64 project) 4.9.2

 Copyright (C) 2014 Free Software Foundation, Inc.
 This is free software; see the source for copying conditions.  There is NO
 warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

(my_cpp_environ)$ cmake --version

> cmake version 3.10

  CMake suite maintained and supported by Kitware (kitware.com/cmake).
```

6. You can deactivate the virtual environment with the *deactivate.bat* script

```
(my_cpp_environ)$ deactivate
```

8.3 Build requirements

There are some requirements that don't feel natural to add to a package recipe. For example, imagine that you had a `cmake/3.4` package in Conan. Would you add it as a requirement to the `ZLib` package, so it will install cmake first in order to build `Zlib`?

In short:

- There are requirements that are only needed when you need to build a package from sources, but if the binary package already exists, you don't want to install or retrieve them.
- These could be dev tools, compilers, build systems, code analyzers, testing libraries, etc.
- They can be very orthogonal to the creation of the package. It doesn't matter whether you build ZLib with CMake 3.4, 3.5 or 3.6. As long as the *CMakeLists.txt* is compatible, it will produce the same final package.
- You don't want to add a lot of different versions (like those of CMake) to be able to use them to build the package. You want to easily change the requirements, without needing to edit the ZLib package recipe.
- Some of them might not be even be taken into account when a package like ZLib is created, such as cross-compiling it to Android (in which the Android toolchain would be a build requirement too).

To address these needs Conan implements `build_requires`.

8.3.1 Declaring build requirements

Build requirements can be declared in profiles, like:

Listing 1: my_profile

```
[build_requires]
Tool1/0.1@user/channel
Tool2/0.1@user/channel, Tool3/0.1@user/channel
*: Tool4/0.1@user/channel
MyPkg*: Tool5/0.1@user/channel
&: Tool6/0.1@user/channel
&!: Tool7/0.1@user/channel
```

Build requirements are specified by a `pattern:`. If such pattern is not specified, it will be assumed to be `*`, i.e. to apply to all packages. Packages can be declared in different lines or by a comma separated list. In this example, `Tool1`, `Tool2`, `Tool3` and `Tool4` will be used for all packages in the dependency graph (while running **`conan install`** or **`conan create`**).

If a pattern like `MyPkg*` is specified, the declared build requirements will only be applied to packages matching that pattern. `Tool5` will not be applied to Zlib for example, but it will be applied to `MyPkgZlib`.

The special case of a **consumer** conanfile (without name or version) it is impossible to match with a pattern, so it is handled with the special character `&`:

- `&` means apply these build requirements to the consumer conanfile
- `&!` means apply the build requirements to all packages except the consumer one.

Remember that the consumer conanfile is the one inside the *test_package* folder or the one referenced in the **`conan install`** command.

Build requirements can be also specified in a package recipe, with the `build_requires` attribute and the `build_requirements()` method:

```
class MyPkg(ConanFile):
    build_requires = "ToolA/0.2@user/testing", "ToolB/0.2@user/testing"

    def build_requirements(self):
        # useful for example for conditional build_requires
        # This means, if we are running on a Windows machine, require ToolWin
        if platform.system() == "Windows":
            self.build_requires("ToolWin/0.1@user/stable")
```

The above `ToolA` and `ToolB` will be always retrieved and used for building this recipe, while the `ToolWin` one will only be used only in Windows.

If some build requirement defined inside `build_requirements()` has the same package name as the one defined in the `build_requires` attribute, the one inside the `build_requirements()` method will prevail.

As a rule of thumb, downstream defined values always override upstream dependency values. If some build requirement is defined in the profile, it will overwrite the build requirements defined in package recipes that have the same package name.

8.3.2 Properties of build requirements

The behavior of `build_requires` is the same irrespective if they are defined in the profile or if defined in the package recipe.

- They will only be retrieved and installed if some package that has to be built from sources and matches the declared pattern. Otherwise, they will not be even checked for existence.
- Options and environment variables declared in the profile as well as in the command line will affect the build requirements for packages. In that way, you can define for example for the `cmake_installer/0.1` package which CMake version will be installed.
- Build requirements will be activated for matching packages via the `deps_cpp_info` and `deps_env_info` members. So, include directories, library names, compile flags (CFLAGS, CXXFLAGS, LINKFLAGS), sysroot, etc. will be applied from the build requirement's package `self.cpp_info` values. The same for `self.env_info`: variables such as `PATH`, `PYTHONPATH`, and any other environment variables will be applied to the matching patterns and activated as environment variables.

- Build requirements can also be transitive. They can declare their own requirements, both normal requirements and their own build requirements. Normal logic for dependency graph resolution applies, such as conflict resolution and dependency overriding.
- Each matching pattern will produce a different dependency graph of build requirements. These graphs are cached so that they are only computed once. If a build requirement applies to different packages with the same configuration it will only be installed once (same behavior as normal dependencies - once they are cached locally, there is no need to retrieve or build them again).
- Build requirements do not affect the binary package ID. If using a different build requirement produces a different binary, you should consider adding an option or a setting to model that (if not already modeled).
- Can also use version-ranges, like `Tool/[>0.3]@user/channel`.
- Build requirements are not listed in **`conan info`** nor are represented in the graph (with **`conan info --graph`**).

8.3.3 Testing libraries

One example of build requirement could be a testing framework, which is implemented as a library. Let's call it `mytest_framework`, an existing Conan package.

Build requirements can be checked for existence (whether they've been applied) in the recipes, which can be useful for conditional logic in the recipes. In this example, we could have one recipe with the following `build()` method:

```
def build(self):
    cmake = CMake(self)
    enable_testing = "mytest_framework" in self.deps_cpp_info.deps
    cmake.configure(defs={"ENABLE_TESTING": enable_testing})
    cmake.build()
    if enable_testing:
        cmake.test()
```

And the package *CMakeLists.txt*:

```
project(PackageTest CXX)
cmake_minimum_required(VERSION 2.8.12)

include(${CMAKE_BINARY_DIR}/conanbuildinfo.cmake)
conan_basic_setup()
if(ENABLE_TESTING)
    add_executable(example test.cpp)
    target_link_libraries(example ${CONAN_LIBS})

    enable_testing()
    add_test(NAME example
             WORKING_DIRECTORY ${CMAKE_BINARY_DIR}/bin
             COMMAND example)
endif()
```

This package recipe will not retrieve the `mytest_framework` nor build the tests, for normal installation:

```
$ conan install .
```

But if the following profile is defined:

Listing 2: mytest_profile

```
[build_requires]
mytest_framework/0.1@user/channel
```

Then the install command will retrieve the `mytest_framework`, build and run the tests:

```
$ conan install . --profile=mytest_profile
```

8.3.4 Common python code

The same technique can be even used to inject and reuse python code in the package recipes, without having to declare dependencies to such python packages.

If a Conan package is defined to wrap and reuse the *mypythontool.py* file:

```
import os
from conans import ConanFile

class Tool(ConanFile):
    name = "PythonTool"
    version = "0.1"
    exports_sources = "mypythontool.py"

    def package(self):
        self.copy("mypythontool.py")

    def package_info(self):
        self.env_info.PYTHONPATH.append(self.package_folder)
```

Then if it is defined in a profile as a build require:

```
[build_requires]
PythonTool/0.1@user/channel
```

such package can be reused in other recipes like this:

```
def build(self):
    self.run("mytool")
    import mypythontool
    self.output.info(mypythontool.hello_world())
```

CHAPTER

NINE

MASTERING CONAN

This section provides an introduction to important productivity features and useful functionalities of conan:

9.1 Use conanfile.py for consumers

You can use a `conanfile.py` for installing/consuming packages, even if you are not creating a package with it. You can also use the existing `conanfile.py` in a given package while developing it to install dependencies, no need to have a separate `conanfile.txt`.

Let's take a look at the complete `conanfile.txt` from the previous *timer* example with POCO library, in which we have added a couple of extra generators

```
[requires]
Poco/1.7.8p3@pocoproject/stable

[generators]
gcc
cmake
txt

[options]
Poco:shared=True
OpenSSL:shared=True

[imports]
bin, *.dll -> ./bin # Copies all dll files from the package "bin" folder to my␣
↪project "bin" folder
lib, *.dylib* -> ./bin # Copies all dylib files from the package "lib" folder to my␣
↪project "bin" folder
```

The equivalent `conanfile.py` file is:

```
from conans import ConanFile, CMake

class PocoTimerConan(ConanFile):
    settings = "os", "compiler", "build_type", "arch"
    requires = "Poco/1.7.8p3@pocoproject/stable" # comma-separated list of requirements
    generators = "cmake", "gcc", "txt"
    default_options = "Poco:shared=True", "OpenSSL:shared=True"

    def imports(self):
        self.copy("*.dll", dst="bin", src="bin") # From bin to bin
        self.copy("*.dylib*", dst="bin", src="lib") # From lib to bin
```

Note that this `conanfile.py` doesn't have a name, version, or `build()` or `package()` method, as it is not creating a package, they are not required.

With this `conanfile.py` you can just work as usual, nothing changes from the user's perspective. You can install the requirements with (from mytimer/build folder):

```
$ conan install ..
```

9.1.1 conan build

One advantage of using `conanfile.py` is that the project build can be further simplified, using the conanfile.py `build()` method.

If you are building your project with CMake, edit your `conanfile.py` and add the following `build()` method:

```
from conans import ConanFile, CMake

class PocoTimerConan(ConanFile):
    settings = "os", "compiler", "build_type", "arch"
    requires = "Poco/1.7.8p3@pocoproject/stable"
    generators = "cmake", "gcc", "txt"
    default_options = "Poco:shared=True", "OpenSSL:shared=True"

    def imports(self):
        self.copy("*.dll", dst="bin", src="bin") # From bin to bin
        self.copy("*.dylib*", dst="bin", src="lib") # From lib to bin

    def build(self):
        cmake = CMake(self)
        cmake.configure()
        cmake.build()
```

Then execute, from your project root:

```
$ conan install . --install-folder build
$ conan build . --build-folder build
```

The **`conan install`** command downloads and prepares the requirements of your project (for the specified settings) and the **`conan build`** command uses all that information to invoke your `build()` method to build your project, which in turn calls `cmake`.

This **`conan build`** will use the settings used in the **`conan install`** which have been cached in the local *conaninfo.txt* and file in your build folder, which simplifies the process and reduces the errors of mismatches between the installed packages and the current project configuration. Also, the *conanbuildinfo.txt* file contains all the needed information obtained from the requirements: `deps_cpp_info`, `deps_env_info`, `deps_user_info` objects.

If you want to build your project for **x86** or another setting just change the parameters passed to **`conan install`**:

```
$ conan install . --install-folder build_x86 -s arch=x86
$ conan build . --build-folder build_x86
```

Implementing and using the conanfile.py `build()` method ensures that we always use the same settings both in the installation of requirements and the build of the project, and simplifies calling the build system.

9.1.2 Other local commands

Conan implements other commands that can be executed locally over a consumer `conanfile.py` which is in user space, not in the local cache:

- **`conan source <path>`**: Execute locally the *conanfile.py* `source()` method.
- **`conan package <path>`**: Execute locally the *conanfile.py* `package()` method.

These commands are mostly used for testing and debugging while developing a new package, before **exporting** such package recipe into the local cache.

See also:

Check the section *Reference/Commands* to find out more.

9.2 Conditional settings, options and requirements

Remember, in your `conanfile.py` you have also access to the options of your dependencies, and you can use them to:

- Add requirements dynamically
- Change values of options

The **configure** method might be used to hardcode dependencies options values. It is strongly discouraged to use it to change the settings values, please remember that `settings` are a configuration *input*, so it doesn't make sense to modify it in the recipes.

Also, for options, a more flexible solution is to define dependencies options values in the `default_options`, not in the `configure()` method, as this would allow to override them. Hardcoding them in the `configure()` method won't allow that and thus won't easily allow conflict resolution. Use it only when it is absolutely necessary that the package dependencies use those options.

Here is an example of what we could do in our **configure method**:

```
...
requires = "Poco/1.9.0@pocoproject/stable" # We will add OpenSSL dynamically "OpenSSL/
↪1.0.2d@lasote/stable"
...

def configure(self):
    # We can control the options of our dependencies based on current options
    self.options["OpenSSL"].shared = self.options.shared

    # Maybe in windows we know that OpenSSL works better as shared (false)
    if self.settings.os == "Windows":
       self.options["OpenSSL"].shared = True

       # Or adjust any other available option
       self.options["Poco"].other_option = "foo"

    # We could check the presence of an option
    if "shared" in self.options:
        pass

def requirements(self):
    # Or add a new requirement!
```

(continues on next page)

(continued from previous page)

```
    if self.options.testing:
        self.requires("OpenSSL/2.1@memsharded/testing")
    else:
        self.requires("OpenSSL/1.0.2d@lasote/stable")
```

See also:

Check the section *Reference/conanfile.py/configure(), config_options()* to find out more.

9.3 Version ranges

Version range expressions are supported, both in `conanfile.txt` and in `conanfile.py` requirements.

The syntax is using brackets. The square brackets are the way to specify conan that is a version range. Otherwise, versions are plain strings, they can be whatever you want them to be (up to limitations of length and allowed characters).

```
class HelloConan(ConanFile):
    requires = "Pkg/[>1.0,<1.8]@user/stable"
```

So when specifying `Pkg/[expression]@user/stable` it means that `expression` will be evaluated as a version range. Otherwise it will be understand as plain text, so `requires = "Pkg/version@user/stable"` always means to use the version `version` literally.

There are some packages that do not follow semver, a popular one would be the OpenSSL package with versions as `1.0.2n`. They cannot be used with version-ranges, to require such packages you always have to use explicit versions (without brackets).

The process to manage plain versions vs version-ranges is also different. The second one requires a "search" in the remote, which is orders of magnitude slower than direct retrieval of the reference (plain versions), so take it into account if you plan to use it for very large projects.

Expressions are those defined and implemented by https://pypi.org/project/node-semver/, but using a comma instead of spaces. Accepted expressions would be:

```
>1.1,<2.1     # In such range
2.8           # equivalent to =2.8
~=3.0         # compatible, according to semver
>1.1 || 0.8   # conditions can be OR'ed
```

Version range expressions are evaluated at the time of building the dependency graph, from downstream to upstream dependencies. No joint-compatibility of the full graph is computed, instead, version ranges are evaluated when dependencies are first retrieved.

This means, that if a package A depends on another package B (A->B), and A has a requirement for `C/[>1.2,<1.8]`, this requirement is evaluated first and it can lead to get the version `C/1.7`. If package B has the requirement to `C/[>1.3,<1.6]`, this one will be overwritten by the downstream one, it will output a version incompatibility error. But the "joint" compatibility of the graph will not be obtained. Downstream packages or consumer projects can impose their own requirements to comply with upstream constraints, in this case a override dependency to `C/[>1.3,<1.6]` can be easily defined in the downstream package or project.

The order of search for matching versions is as follows:

- First, the local conan storage is searched for matching versions, unless the **`--update`** flag is provided to **`conan install`**.
- If a matching version is found, it is used in the dependency graph as a solution.

- If no matching version is locally found, it starts to search in the remotes, in order. If some remote is specified with **-r=remote**, then only that remote will be used.
- If the **--update** parameter is used, then the existing packages in the local conan cache will not be used, and the same search of the previous steps is carried out in the remotes. If new matching versions are found, they will be retrieved, so subsequent calls to **install** will find them locally and use them.

9.4 Build policies

By default, **conan install** command will search for a binary package (corresponding to our settings and defined options) in a remote, if it's not present the install command will fail.

As previously demonstrated, we can use the **–build** option to change the default **conan install** behaviour:

- **--build some_package** will build only "some_package".
- **--build missing** will build only the missing requires.
- **--build** will build all requirements from sources.
- **--build outdated** will try to build from code if the binary is not built with the current recipe or when missing binary package.

With the `build_policy` attribute the package creator can change the default conan's build behavior. The allowed build_policy values are:

- `missing`: If no binary package is found, conan will build it without the need of invoke conan install with **--build missing** option.
- `always`: The package will be built always, **retrieving each time the source code** executing the "source" method.

```
class PocoTimerConan(ConanFile):
    settings = "os", "compiler", "build_type", "arch"
    requires = "Poco/1.7.8p3@pocoproject/stable" # comma-separated list of␣
↪requirements
    generators = "cmake", "gcc", "txt"
    default_options = "Poco:shared=True", "OpenSSL:shared=True"
    build_policy = "always" # "missing"
```

These build policies are especially useful if the package creator doesn't want to provide binary package, for example, with header only libraries.

The `always` policy, will retrieve the sources each time the package is installed so it can be useful for providing a "latest" mechanism or ignoring the uploaded binary packages.

9.5 Environment variables

There are several use cases for environment variables:

- Conan global configuration environment variables (e.g. `CONAN_COMPRESSION_LEVEL`). They can be configured in *conan.conf* or as system environment variables, and control conan behavior.
- Package recipes can access environment variables to determine their behavior. A typical example would be when launching CMake, it will check for CC and CXX environment variables to define the compiler to use. These variables are mostly transparent for conan, and just used by the package recipes.
- **Environment variables can be set in different ways:**

- global, at the OS level, with **`export VAR=Value`** or in Windows **`SET VAR=Value`**.
- In the conan command line: **`conan install -e VAR=Value`**.
- In profile files.
- In package recipes in the `self.env_info` field, so they are activated for dependent recipes.

9.5.1 Defining environment variables

You can use *profiles* to define environment variables that will apply to your recipes. You can also use **`-e`** parameter in **`conan install`**, **`conan info`** and **`conan create`** commands.

```
[env]
CC=/usr/bin/clang
CXX=/usr/bin/clang++
```

If you want to override an environment variable that a package has inherited from its requirements, you can use either **profiles** or **`-e`** to do it:

```
conan install -e MyPackage:PATH=/other/path
```

If you want to define an environment variable but you want to append the variables declared in your requirements you can use the `[]` syntax:

```
$ conan install -e PYTHONPATH=[/other/path]
```

This way the first entry in the PYTHONPATH variable will be */other/path* but the PYTHONPATH values declared in the requirements of the project will be appended at the end using the system path separator.

9.5.2 Automatic environment variables inheritance

If your dependencies define some `env_info` variables in the `package_info()` method they will be automatically applied before calling the consumer *conanfile.py* methods `source()`, `build()`, `package()` and `imports()`. You can read more about `env_info` object *here*.

For example, if you are creating a package for a tool, you can define the variable `PATH`:

```
class ToolExampleConan(ConanFile):
    name = "my_tool_installer"
    ...

    def package_info(self):
        self.env_info.path.append(os.path.join(self.package_folder, "bin"))
```

If another conan recipe requires the *my_tool_installer* in the `source()`, `build()`, `package()` and `imports()` the bin folder of the `my_tool_installer` package will be automatically appended to the system PATH. If `my_tool_installer` packages an executable called `my_tool_executable` in the *bin* of the package folder we can directly call the tool, because it will be available in the path:

```
class MyLibExample(ConanFile):
    name = "my_lib_example"
    ...

    def build(self):
        self.run("my_tool_executable some_arguments")
```

You could also set `CC`, `CXX` variables if we are packing a compiler to define what compiler to use or any other environment variable. Read more about tool packages *here*.

9.6 Virtual Environments

Conan offer three special conan generators to create virtual environments:

- `virtualenv`: Declares the *self.env_info* variables of the requirements.
- `virtualbuildenv`: Special build environment variables for autotools/visual studio.
- `virtualrunenv`: Special environment variables to locate executables and shared libraries in the requirements.

These virtual environment generators create two executable script files (.sh or .bat depending on the current operating system), one for `activate` the virtual environment (set the environment variables) and one for `deactivate` it.

You can aggregate two or more virtual environments, that means that you can activate a `virtualenv` and then activate a `virtualrunenv` so you will have available the environment variables declared in the `env_info` object of the requirements plus the special enviroment variables to locate executables and shared libraries.

9.6.1 Virtualenv generator

Conan provides a **virtualenv** generator, able to read from each dependency the *self.env_info* variables declared in the `package_info()` method and generate two scripts "activate" and "deactivate". These scripts set/unset all env variables in the current shell.

Example:

The recipe of `cmake_installer/3.9.0@conan/stable` appends to the PATH variable the package folder/bin.

You can check existing CMake conan package versions in conan-center with:

```
$ conan search cmake* -r=conan-center
```

In the **bin** folder there is a **cmake** executable:

```
def package_info(self):
  self.env_info.path.append(os.path.join(self.package_folder, "bin"))
```

Let's prepare a virtual environment to have available our cmake in the path, open `conanfile.txt` and change (or add) **virtualenv** generator:

```
[requires]
cmake_installer/3.9.0@conan/stable

[generators]
virtualenv
```

Run **`conan install`**:

```
$ conan install .
```

You can also avoid the creation of the *conanfile.txt* completely and directly do:

```
$ conan install cmake_installer/3.9.0@conan/stable -g=virtualenv
```

And activate the virtual environment, and now you can run `cmake --version` and check that you have the installed CMake in path.

```
$ source activate.sh # Windows: activate.bat without the source
$ cmake --version
```

Two sets of scripts are available for Windows - `activate.bat`/`deactivate.bat` and `activate.ps1`/`deactivate.ps1` if you are using powershell. Deactivate the virtual environment (or close the console) to restore the environment variables:

```
$ source deactivate.sh # Windows: deactivate.bat without the source
```

See also:

Read the Howto *Create installer packages* to know more about virtual environment feature. Check the section *Reference/virtualenv* to see the reference of the generator.

9.6.2 Virtualbuildenv environment

Use the generator `virtualbuildenv` to activate an environment that will set the environment variables for Autotools and Visual Studio.

The generator will create `activate_build` and `deactivate_build` files.

See also:

Read More about the building environment variables defined in the sections *Building with autotools* and *Build with Visual Studio*.

Check the section *Reference/virtualbuildenv* to see the reference of the generator.

9.6.3 Virtualrunenv generator

Use the generator `virtualrunenv` to activate an environment that will:

- Append to `PATH` environment variable every `bin` folder of your requirements.
- Append to `LD_LIBRARY_PATH` and `DYLD_LIBRARY_PATH` environment variables each `lib` folder of your requirements.

The generator will create `activate_run` and `deactivate_run` files. This generator is especially useful:

- If you are requiring packages with shared libraries and you are running some executable that needs those libraries.
- If you have a requirement with some tool (executable) and you need it in the path.

In the previous example of the `cmake_installer` recipe, even if the cmake_installer package doesn't declare the `self.env_info.path` variable, using the virtualrunenv generator, the `bin` folder of the package will be available in the PATH. So after activating the virtual environment we could just run `cmake` and we will be executing the cmake of the package.

See also:

- *Reference/Tools/environment_append*

9.7 Logging

9.7.1 How to log and debug a conan execution

You can use the *CONAN_TRACE_FILE* environment variable to log and debug several conan command execution. Set the `CONAN_TRACE_FILE` environment variable pointing to a log file.

Example:

```
export CONAN_TRACE_FILE=/tmp/conan_trace.log # Or SET in windows
conan install zlib/1.2.8@lasote/stable
```

The */tmp/conan_trace.log* file:


```
{"_action": "COMMAND", "name": "install", "parameters": {"all": false, "build": null,
↪"env": null, "file": null, "generator": null, "manifests": null, "manifests_
↪interactive": null, "no_imports": false, "options": null, "package": null, "profile
↪": null, "reference": "zlib/1.2.8@lasote/stable", "remote": null, "scope": null,
↪"settings": null, "update": false, "verify": null, "werror": false}, "time":␣
↪1485345289.250117}
{"_action": "REST_API_CALL", "duration": 1.8255090713500977, "headers": {
↪"Authorization": "**********", "X-Client-Anonymous-Id": "**********", "X-Client-Id
↪": "lasote2", "X-Conan-Client-Version": "0.19.0-dev"}, "method": "GET", "time":␣
↪1485345291.092218, "url": "https://server.conan.io/v1/conans/zlib/1.2.8/lasote/
↪stable/download_urls"}
{"_action": "DOWNLOAD", "duration": 0.4136989116668701, "time": 1485345291.506399,
↪"url": "https://conanio-production.s3.amazonaws.com/storage/zlib/1.2.8/lasote/
↪stable/export/conanmanifest.txt"}
{"_action": "DOWNLOAD", "duration": 0.10367798805236816, "time": 1485345291.610335,
↪"url": "https://conanio-production.s3.amazonaws.com/storage/zlib/1.2.8/lasote/
↪stable/export/conanfile.py"}
{"_action": "DOWNLOAD", "duration": 0.059114933013916016, "time": 1485345291.669744,
↪"url": "https://conanio-production.s3.amazonaws.com/storage/zlib/1.2.8/lasote/
↪stable/export/conan_export.tgz"}
{"_action": "DOWNLOADED_RECIPE", "_id": "zlib/1.2.8@lasote/stable", "duration": 2.
↪40762996673584, "files": {"conan_export.tgz": "/home/laso/.conan/data/zlib/1.2.8/
↪lasote/stable/export/conan_export.tgz", "conanfile.py": "/home/laso/.conan/data/
↪zlib/1.2.8/lasote/stable/export/conanfile.py", "conanmanifest.txt": "/home/laso/.
↪conan/data/zlib/1.2.8/lasote/stable/export/conanmanifest.txt"}, "remote": "conan.io
↪", "time": 1485345291.670017}
{"_action": "REST_API_CALL", "duration": 0.4844989776611328, "headers": {
↪"Authorization": "**********", "X-Client-Anonymous-Id": "**********", "X-Client-Id
↪": "lasote2", "X-Conan-Client-Version": "0.19.0-dev"}, "method": "GET", "time":␣
↪1485345292.160912, "url": "https://server.conan.io/v1/conans/zlib/1.2.8/lasote/
↪stable/packages/c6d75a933080ca17eb7f076813e7fb21aaa740f2/download_urls"}
{"_action": "DOWNLOAD", "duration": 0.06388187408447266, "time": 1485345292.225308,
↪"url": "https://conanio-production.s3.amazonaws.com/storage/zlib/1.2.8/lasote/
↪stable/package/c6d75a933080ca17eb7f076813e7fb21aaa740f2/conaninfo.txt?
↪Signature=c1KAOqvxtCUnnQOeYizZ9bgcwwY%3D&Expires=1485352492&
↪AWSAccessKeyId=AKIAJXMWDMVCDMAZQK5Q"}
{"_action": "REST_API_CALL", "duration": 0.8182470798492432, "headers": {
↪"Authorization": "**********", "X-Client-Anonymous-Id": "**********", "X-Client-Id
↪": "lasote2", "X-Conan-Client-Version": "0.19.0-dev"}, "method": "GET", "time":␣
↪1485345293.044904, "url": "https://server.conan.io/v1/conans/zlib/1.2.8/lasote/
↪stable/packages/c6d75a933080ca17eb7f076813e7fb21aaa740f2/download_urls"}
{"_action": "DOWNLOAD", "duration": 0.07849907875061035, "time": 1485345293.123831,
↪"url": "https://conanio-production.s3.amazonaws.com/storage/zlib/1.2.8/lasote/
↪stable/package/c6d75a933080ca17eb7f076813e7fb21aaa740f2/conanmanifest.txt"}
```


(continues on next page)

(continued from previous page)


```
{"_action": "DOWNLOAD", "duration": 0.06638002395629883, "time": 1485345293.190465,
↪"url": "https://conanio-production.s3.amazonaws.com/storage/zlib/1.2.8/lasote/
↪stable/package/c6d75a933080ca17eb7f076813e7fb21aaa740f2/conaninfo.txt"}
{"_action": "DOWNLOAD", "duration": 0.3634459972381592, "time": 1485345293.554206,
↪"url": "https://conanio-production.s3.amazonaws.com/storage/zlib/1.2.8/lasote/
↪stable/package/c6d75a933080ca17eb7f076813e7fb21aaa740f2/conan_package.tgz"}
{"_action": "DOWNLOADED_PACKAGE", "_id": "zlib/1.2.8@lasote/
↪stable:c6d75a933080ca17eb7f076813e7fb21aaa740f2", "duration": 1.3279249668121338,
↪"files": {"conan_package.tgz": "/home/laso/.conan/data/zlib/1.2.8/lasote/stable/
↪package/c6d75a933080ca17eb7f076813e7fb21aaa740f2/conan_package.tgz", "conaninfo.txt
↪": "/home/laso/.conan/data/zlib/1.2.8/lasote/stable/package/
↪c6d75a933080ca17eb7f076813e7fb21aaa740f2/conaninfo.txt", "conanmanifest.txt": "/
↪home/laso/.conan/data/zlib/1.2.8/lasote/stable/package/
↪c6d75a933080ca17eb7f076813e7fb21aaa740f2/conanmanifest.txt"}, "remote": "conan.io",
↪"time": 1485345293.554466}
```


In the traces we can see:

1. A command `install` execution.
2. A rest api call to get some `download_urls`.
3. Three files downloaded (corresponding to the previously retrieved urls).
4. `DOWNLOADED_RECIPE` tells us that the recipe retrieving is finished. We can see that the whole retrieve process took 2.4 seconds.
5. conan client has computed the needed binary package SHA and now will get it. So will request and download the package `package_id` file to perform some checks like outdated binaries.
6. Another rest api call to get some more `download_urls`, for the package files and download them.
7. Finally we get a `DOWNLOADED_PACKAGE` telling us that the package has beed downloaded. It took 1.3 seconds.

If we execute conan install again:

```
export CONAN_TRACE_FILE=/tmp/conan_trace.log # Or SET in windows
conan install zlib/1.2.8@lasote/stable
```

The */tmp/conan_trace.log* file only three lines will be appended:


```
{"_action": "COMMAND", "name": "install", "parameters": {"all": false, "build": null,
↪"env": null, "file": null, "generator": null, "manifests": null, "manifests_
↪interactive": null, "no_imports": false, "options": null, "package": null, "profile
↪": null, "reference": "zlib/1.2.8@lasote/stable", "remote": null, "scope": null,
↪"settings": null, "update": false, "verify": null, "werror": false}, "time":␣
↪1485346039.817543}
{"_action": "GOT_RECIPE_FROM_LOCAL_CACHE", "_id": "zlib/1.2.8@lasote/stable", "time":␣
↪1485346039.824949}
{"_action": "GOT_PACKAGE_FROM_LOCAL_CACHE", "_id": "zlib/1.2.8@lasote/
↪stable:c6d75a933080ca17eb7f076813e7fb21aaa740f2", "time": 1485346039.827915}
```


1. A command `install` execution.
2. A `GOT_RECIPE_FROM_LOCAL_CACHE` because we already have it available in local cache.
3. A `GOT_PACKAGE_FROM_LOCAL_CACHE` because the package is cached too.

9.7.2 How to log the build process

You can log your command executions `self.run` in a file named `conan_run.log` using the environment variable CONAN_LOG_RUN_FILE. Check for more details here: *CONAN_LOG_RUN_TO_FILE*.

You can also use the variable *CONAN_PRINT_RUN_COMMANDS* to log extra information about the commands being executed.

Package the log files

The *conan_run.log'* file will be available in your `build` folder so you can package it the same way you package a library file:

```
def package(self):
    self.copy(pattern="conan_run.log", dst="", keep_path=False)
```

9.8 Sharing the settings and other configuration

If you are using Conan in a company or in an organization, sometimes you need to share the *settings.yml* file or the *profiles*, or even the *remotes* or any other conan local configuration with the team.

You can use the **`conan config install`**.

If you want to try this feature without affecting to your current configuration you can declare the `CONAN_USER_HOME` environment variable and point to a different directory.

Read more in the section *reference/commands/conan config install*.

9.9 Conan local cache: concurrency, Continuous Integration, isolation

Conan needs access to some, per user, configuration files, as the **conan.conf** file that defines the basic client app configuration. By convention, this file will be located in the user home folder **~/.conan/**. This folder will typically also store the package cache, in **~/.conan/data**. Though the latter is configurable in *conan.conf*, still conan needs some place to look for this initial configuration file.

There are some scenarios in which you might want to use different initial locations for the conan client application:

- Continuous Integration (CI) environments, in which multiple jobs can also work concurrently. Moreover, these environments would typically want to run with different user credentials, different remote configurations, etc. Note that using Continuous Integration with the same user, with isolated machine instances (virtual machines), or with sequential jobs is perfectly possible. For example, we use a lot CI cloud services of travis-ci and appveyor.
- Independent per project management and storage. If as a single developer you want to manage different projects with different user credentials (for the same remote, having different users for different remotes is also fine), consuming packages from different remotes, you might find that having a single user configuration is not enough. Having independent caches might allow also to take away with you very easily the requirements of a certain project.

Using different caches is very simple. You can just define the environment variable **CONAN_USER_HOME**. By setting this variable to different paths, you have multiple conan caches, something like python "virtualenvs". Just

changing the value of **CONAN_USER_HOME** you can switch among isolated conan instantes that will have independent package storage caches, but also different user credentials, different user default settings, and different remotes configuration.

Note: Use an absolute path or a path starting with ~/ (relative to user home). In Windows do not use quotes.

Windows users:

```
$ SET CONAN_USER_HOME=c:\data
$ conan install . # call conan normally, config & data will be in c:\data
```

Linux/OSx users:

```
$ export CONAN_USER_HOME=/tmp/conan
$ conan install . # call conan normally, config & data will be in /tmp/conan
```

You can now:

- Build concurrent jobs, parallel builds in Continous Integration or locally, just setting the variable before launching conan commands.
- You can test locally different user credentials, default configurations, different remotes, just by switching from one cache to the others.

```
$ export CONAN_USER_HOME=/tmp/conan
$ conan search  # using that /tmp/conan cache
$ conan user  # using that /tmp/conan cache

$ export CONAN_USER_HOME=/tmp/conan2
$ conan search  # different packages
$ conan user  # can be different users

$ export CONAN_USER_HOME=/tmp/conan  # just go back to use the other cache
```

9.9.1 Concurrency

Conan local cache support some degree of concurrency, allowing simultaneous creation or installation of different packages, or building different binaries for the same package. However, concurrent operations like removal of packages while creating them will fail. If you need different environments that operate totally independently, you probably want to use different conan caches for that.

The concurrency is implemented with a Readers-Writers lock mechanism, which in turn uses `fasteners` library file locks to achieve multi-platform portability. As this "mutex" resource is by definition not enough to implement a Readers-Writers solution, some active-wait with time sleeps in a loop is necessary. However, this time sleeps will be rare, only sleeping when there is actually a collision and waiting on a lock.

The lock files will be stored inside each `Pkg/version/user/channel` folder in the local cache, in a `rw` file for locking the entire package, or in a set of locks (one per each different binary package, under a subfolder called `locks`, each lock named with the binary ID of the package).

It is possible to disable the locking mechanism in `conan.conf`:

```
[general]
cache_no_locks = True
```

CHAPTER

TEN

SYSTEMS AND CROSS BUILDING

This section explains how to cross build with Conan to any platform and the Windows subsystems (Cygwin, MSYS2).

10.1 Cross building

Cross building is compiling a library or executable in one platform to be used in a different one.

Cross-compilation is used to build software for embedded devices where you don't have an operating system nor a compiler available. Also for building software for not too fast devices, like an Android machine, a Raspberry PI etc.

To cross build code you need the right toolchain. A toolchain is basically a compiler with a set of libraries matching the `host` platform.

10.1.1 GNU triplet convention

According to the GNU convention, there are three platforms involved in the software building:

- **Build platform:** The platform on which the compilation tools are executed
- **Host platform:** The platform on which the code will run
- **Target platform:** Only when building a compiler, this is the platform that the compiler will generate code for

When you are building code for your own machine it's called **native building**, where the `build` and the `host` platforms are the same. The `target` platform is not defined in this situation.

When you are building code for a different platform, it's called **cross building**, where the `build` platform is different from the `host` platform. The `target` platform is not defined in this situation.

The use of the `target` platform is rarely needed, only makes sense when you are building a compiler. For instance, when you are building in your Linux machine a GCC compiler that will run on Windows, to generate code for Android. Here, the `build` is your Linux computer, the `host` is the Windows computer and the `target` is Android.

10.1.2 Conan settings

From version 1.0, Conan introduces new settings to model the GNU convention triplet:

`build` platform settings:

- **os_build**: Operating system of the `build` system.

- **arch_build**: Architecture system of the `build` system.

These settings are detected the first time you run Conan with the same values than the `host` settings, so by default, we are doing **native building**. Probably you will never need to change the value of this settings because they describe where are you running Conan.

`host` platform settings:

- **os**: Operating system of the `host` system.
- **arch**: Architecture of the `host` system.
- **compiler**: Compiler of the `host` system (to declare compatibility of libs in the host platform)
- ... (all the regular settings)

These settings are the regular Conan settings, already present before supporting the GNU triplet convention. If you are cross building you have to change them according to the `host` platform.

`target` platform:

- **os_target**: Operating system of the `target` system.
- **arch_target**: Architecture of the `target` system.

If you are building a compiler, specify with these settings where the compiled code will run.

10.1.3 Cross building with Conan

If you want to cross-build a Conan package, for example, in your Linux machine, build the *zlib* Conan package for Windows, you need to indicate to Conan where to find your cross-compiler/toolchain.

There are two approaches:

- Install the toolchain in your computer and use a `profile` to declare the settings and point to the needed tools/libraries in the toolchain using the `[env]` section to declare, at least, the `CC` and `CXX` environment variables.
- Package the toolchain as a Conan package and include it as a `build_require`.

Using profiles

Create a profile with:

- A **[settings]** section containing the needed settings: `os_build`, `arch_build` and the regular settings `os`, `arch`, `compiler`, `build_type` and so on.
- An **[env]** section with a PATH variable pointing to your installed toolchain. Also any other variable that the toolchain expects (read the docs of your compiler). Some build systems need a variable `SYSROOT` to locate where the host system libraries and tools are.

Linux to Windows

- Install the needed toolchain, in ubuntu:

```
sudo apt-get install g++-mingw-w64 gcc-mingw-w64
```

- Create a file named **linux_to_win64** with the contents:

```
[env]
# Where is our C compiler
CC=/usr/bin/x86_64-w64-mingw32-gcc
# Where is our CPP compiler
CXX=/usr/bin/x86_64-w64-mingw32-g++

[settings]
# We are building in Ubuntu Linux
os_build=Linux
arch_build=x86_64

# We are cross building to Windows
os=Windows
arch=x86_64
compiler=gcc
# Adjust to the gcc version of your MinGW package
compiler.version=6.3
compiler.libcxx=libstdc++11
build_type=Release
```

- Clone an example recipe or use your own recipe:

```
git clone https://github.com/memsharded/conan-hello.git
```

- Call **conan create** using the created profile.win

```
$ cd conan-hello && conan create . conan/testing --profile ../linux_to_win64
...
[ 50%] Building CXX object CMakeFiles/example.dir/example.cpp.obj
[100%] Linking CXX executable bin/example.exe
[100%] Built target example
```

A *bin/example.exe* for Win64 platform has been built.

Windows to Raspberry PI (Linux/ARM)

- Install the toolchain: http://gnutoolchains.com/raspberry/ You can choose different versions of the GCC cross compiler, choose one and adjust the following settings in the profile accordingly.
- Create a file named **win_to_rpi** with the contents:

```
target_host=arm-linux-gnueabihf
standalone_toolchain=C:/sysgcc/raspberry
cc_compiler=gcc
cxx_compiler=g++

[settings]
os_build=Windows
arch_build=x86_64
os=Linux
arch=armv7 # Change to armv6 if you are using Raspberry 1
compiler=gcc
compiler.version=6
compiler.libcxx=libstdc++11
build_type=Release
```

(continues on next page)

(continued from previous page)

```
[env]
CONAN_CMAKE_FIND_ROOT_PATH=$standalone_toolchain/$target_host/sysroot
PATH=[$standalone_toolchain/bin]
CHOST=$target_host
AR=$target_host-ar
AS=$target_host-as
RANLIB=$target_host-ranlib
LD=$target_host-ld
STRIP=$target_host-strip
CC=$target_host-$cc_compiler
CXX=$target_host-$cxx_compiler
CXXFLAGS=-I"$standalone_toolchain/$target_host/lib/include"
```

The profiles to target Linux are all very similar, probably you just need to adjust the variables declared in the top of the profile:

- **target_host**: All the executables in the toolchain starts with this prefix.
- **standalone_toolchain**: Path to the toolchain installation.
- **cc_compiler/cxx_compiler**: In this case `gcc`/`g++`, but could be `clang`/`clang++`.
- Clone an example recipe or use your own recipe:

```
git clone https://github.com/memsharded/conan-hello.git
```

- Call **`conan create`** using the created profile.

```
$ cd conan-hello && conan create . conan/testing --profile=../win_to_rpi
...
[ 50%] Building CXX object CMakeFiles/example.dir/example.cpp.obj
[100%] Linking CXX executable bin/example
[100%] Built target example
```

A *bin/example* for Raspberry PI (Linux/armv7hf) platform has been built.

Linux/Windows/Macos to Android

Cross bulding a library for Android is very similar to the previous examples, except the complexity of managing different architectures (armeabi, armeabi-v7a, x86, arm64-v8a) and the Android API levels.

Download the Android NDK here and unzip it.

Note: If you are in Windows the process will be almost the same, but unzip the file in the root folder of your hard disk (C:) to avoid issues with path lengths.

Now you have to build a standalone toolchain, we are going to target "arm" architecture and the Android API level 21, change the `--install-dir` to any other place that works for you:

```
$ cd build/tools
$ python make_standalone_toolchain.py --arch=arm --api=21 --stl=libc++ --install-dir=/
↪myfolder/arm_21_toolchain
```

Note: You can generate the standalone toolchain with several different options to target different architectures, api levels etc.

Check the Android docs: standalone toolchain

To use the `clang` compiler, create a profile `android_21_arm_clang`. Once again, the profile is very similar to the RPI one:

```
standalone_toolchain=/myfolder/arm_21_toolchain # Adjust this path
target_host=arm-linux-androideabi
cc_compiler=clang
cxx_compiler=clang++

[settings]
compiler=clang
compiler.version=5.0
compiler.libcxx=libc++
os=Android
os.api_level=21
arch=armv7
build_type=Release

[env]
CONAN_CMAKE_FIND_ROOT_PATH=$standalone_toolchain/sysroot
PATH=[$standalone_toolchain/bin]
CHOST=$target_host
AR=$target_host-ar
AS=$target_host-as
RANLIB=$target_host-ranlib
CC=$target_host-$cc_compiler
CXX=$target_host-$cxx_compiler
LD=$target_host-ld
STRIP=$target_host-strip
CFLAGS= -fPIE -fPIC -I$standalone_toolchain/include/c++/4.9.x
CXXFLAGS= -fPIE -fPIC -I$standalone_toolchain/include/c++/4.9.x
LDFLAGS= -pie
```

You could also use `gcc` using this profile `arm_21_toolchain_gcc`, changing the `cc_compiler` and `cxx_compiler` variables, removing `-fPIE` flag and, of course, changing the `[settings]` to match the gcc toolchain compiler:

```
standalone_toolchain=/myfolder/arm_21_toolchain
target_host=arm-linux-androideabi
cc_compiler=gcc
cxx_compiler=g++

[settings]
compiler=gcc
compiler.version=4.9
compiler.libcxx=libstdc++
os=Android
os.api_level=21
arch=armv7
build_type=Release

[env]
CONAN_CMAKE_FIND_ROOT_PATH=$standalone_toolchain/sysroot
PATH=[$standalone_toolchain/bin]
CHOST=$target_host
AR=$target_host-ar
```

(continues on next page)

(continued from previous page)

```
AS=$target_host-as
RANLIB=$target_host-ranlib
CC=$target_host-$cc_compiler
CXX=$target_host-$cxx_compiler
LD=$target_host-ld
STRIP=$target_host-strip
CFLAGS= -fPIC -I$standalone_toolchain/include/c++/4.9.x
CXXFLAGS= -fPIC -I$standalone_toolchain/include/c++/4.9.x
LDFLAGS=
```

- Clone, for example, the zlib library to try to build it to Android

```
git clone https://github.com/lasote/conan-zlib.git
```

- Call **conan create** using the created profile.

```
$ cd conan-zlib && conan create . conan/testing --profile=../android_21_arm_clang

...
-- Build files have been written to: /tmp/conan-zlib/test_package/build/
↪ba0b9dbae0576b9a23ce7005180b00e4fdef1198
Scanning dependencies of target enough
[ 50%] Building C object CMakeFiles/enough.dir/enough.c.o
[100%] Linking C executable bin/enough
[100%] Built target enough
zlib/1.2.11@conan/testing (test package): Running test()
```

A **bin/enough** for Android ARM platform has been built.

Using build requires

Instead of downloading manually the toolchain and creating a profile, you can create a Conan package with it. The toolchain Conan package needs to fill the `env_info` object in the *package_info()* method with the same variables we've specified in the examples above in the `[env]` section of profiles.

A layout of a Conan package for a toolchain could looks like this:

```
from conans import ConanFile
import os


class MyToolchainXXXConan(ConanFile):
    name = "my_toolchain"
    version = "0.1"
    settings = "os_build", "arch_build"

    def build(self):
        # Typically download the toolchain for the 'build' host
        url = "http://fake_url.com/installers/%s/%s/toolchain.tgz" % (os_build, os_
↪arch)
        tools.download(url, "toolchain.tgz")
        tools.unzip("toolchain.tgz")

    def package(self):
        # Copy all the
        self.copy("*", dst="", src="toolchain")
```

(continues on next page)

(continued from previous page)

```
    def package_info(self):
        bin_folder = os.path.join(self.package_folder, "bin")
        self.env_info.path.append(bin_folder)
        self.env_info.CC = os.path.join(bin_folder, "mycompiler-cc")
        self.env_info.CXX = os.path.join(bin_folder, "mycompiler-cxx")
        self.env_info.SYSROOT = self.package_folder
```

Finally, when you want to cross-build a library, the profile to be used, will include a `[build_requires]` section with the reference to our new packaged toolchain. Also will contain a `[settings]` section with the same settings of the examples above.

You can

See also:

- Check the *Creating conan packages to install dev tools* to learn more about how to create Conan packages for tools.
- Check the mingw-installer build require recipe as an example of packaging a compiler.

Using Docker images

You can use some *available docker images with Conan preinstalled images* to cross build conan packages. Currently there are `i386`, `armv7` and `armv7hf` images with the needed packages and toolchains installed to cross build.

Example: Cross-building and uploading a package along with all its missing dependencies for `Linux/armv7hf` is done in few steps:

```
$ git clone https://github.com/conan-community/conan-openssl
$ cd conan-openssl
$ docker run -it -v$(pwd):/home/conan/project --rm lasote/conangcc49-armv7hf /bin/bash

# Now we are running on the conangcc49-armv7hf container
$ sudo pip install conan --upgrade
$ cd project

$ conan create . user/channel --build missing
$ conan remote add myremoteARMV7 http://some.remote.url
$ conan upload "*" -r myremoteARMV7 --all
```

Check the section: *How to run Conan with Docker* to know more.

Preparing recipes to be cross-compiled

If you use the build helpers *AutoToolsBuildEnvironment* or *CMake*, Conan will adjust the configuration accordingly to the specified settings.

If don't, you can always check the `self.settings.os`, `self.settings.build_os`, `self.settings.arch` and `self.settings.build_arch` settings values and inject the needed flags to your build system script.

You can use this tool to check if you are cross building:

- *tools.cross_building(self.settings)* (returns True or False)

10.1.4 ARM architecture reference

Remember that the conan settings are intended to unify the different names for operating systems, compilers, architectures etc.

Conan has different architecture settings for ARM: `armv6`, `armv7`, `armv7hf`, `armv8`. The "problem" with ARM architecture is that frequently are named in different ways, so maybe you are wondering what setting do you need to specify in your case.

Here is a table with some typical ARM platorms:

Platform	Conan setting
Raspberry PI 1	`armv6`
Raspberry PI 2	`armv7` or `armv7hf` if we want to use the float point hard support
Raspberry PI 3	`armv8` also known as armv64-v8a
Visual Studio	`armv7` currently Visual Studio builds `armv7` binaries when you select ARM.
Android armbeabi-v7a	`armv7`
Android armv64-v8a	`armv8`
Android armeabi	`armv6` (as a minimal compatible, will be compatible with v7 too)

See also:

Reference links

ARM

- https://developer.arm.com/docs/dui0773/latest/compiling-c-and-c-code/specifying-a-target-architecture-processor-and-instructio
- https://developer.arm.com/docs/dui0774/latest/compiler-command-line-options/-target
- https://developer.arm.com/docs/dui0774/latest/compiler-command-line-options/-march

ANDROID

- https://developer.android.com/ndk/guides/standalone_toolchain

VISUAL STUDIO

- https://msdn.microsoft.com/en-us/library/dn736986.aspx

See also:

- See *conan.conf file* and *Environment variables* sections to know more.
- See *AutoToolsBuildEnvironment build helper* reference.
- See *CMake build helper* reference.
- See CMake cross building wiki to know more about cross building with CMake.

10.2 Windows Subsystems

On Windows, you can run different *subsystems* that enhance with UNIX capabilities the operating system.

Conan supports `MSYS2`, `CYGWIN`, `WSL` and in general any subsystem that is able to run a `bash` terminal.

Many libraries use these subsystems to be able to use the Unix tools like the `Autoconf` suite to generate and build `Makefiles`.

The difference between MSYS2 and CYGWIN is that MSYS2 is oriented to the development of native Windows packages, while CYGWIN tries to provide a complete unix-like system to run any Unix application on it.

For that reason, we recommend the use of MSYS2 as a subsystem to be used with Conan.

10.2.1 Operation Modes

The `MSYS2` and `CYGWIN` can be used with different operation modes:

- You can use them together with MinGW to build Windows-native software.
- You can use them together with any other compiler to build Windows-native software, even with Visual Studio.
- You can use them with MinGW to build specific software for the subsystem, with a dependency to a runtime DLL (`msys-2.0.dll` and `cygwin1.dll`)

If you are building specific software for the subsystem, you have to specify a value for the setting `os.subsystem`, if you are only using the subsystem for taking benefit of the UNIX tools but generating native Windows software, you shouldn't specify it.

10.2.2 Running commands inside the subsystem

self.run()

In a Conan recipe, you can use the `self.run` method specifying the parameter `win_bash=True` that will call automatically to the tool *tools.run_in_windows_bash*.

It will use the **bash** in the path or the **bash** specified for the environment variable *CONAN_BASH_PATH* to run the specified command.

Conan will automatically escape the command to match the detected subsystem. If you also specify the `msys_mingw` parameter to False, and the subsystem is `MSYS2` it will run in Windows-native mode, the compiler won't link against the `msys-2.0.dll`.

AutoToolsBuildEnvironment

In the constructor of the build helper, you have the `win_bash` parameter. Set it to `True` to run the `configure` and `make` commands inside a bash.

10.2.3 Controlling the build environment

Building software in a Windows subsystem for a different compiler than MinGW can be painful sometimes. The reason is how the subsystem finds your compiler/tools in your system.

For example, the icu library requires Visual Studio to be built in Windows, but also a subsystem able to build the Makefile. A very common problem and example of the pain is the `link.exe` program. In the Visual Studio suite, `link.exe` is the linker, but in the `MSYS2` environment the `link.exe` is a tool to manage symbolic links.

Conan is able to prioritize the tools when you use `build_requires`, and put the tools in the PATH in the right order.

There are some packages you can use as `build_requires`:

- From Conan-center:
  - **mingw_installer/1.0@conan/stable**: MinGW compiler installer as a Conan package.
  - **msys2_installer/latest@bincrafters/stable**: MSYS2 subsystem as a Conan package.
  - **cygwin_installer/2.9.0@bincrafters/stable**: Cygwin subsystem as a Conan package.

For example, create a profile and name it *msys2_mingw* with the following contents:

```
[build_requires]
mingw_installer/1.0@conan/stable
msys2_installer/latest@bincrafters/stable

[settings]
os_build=Windows
os=Windows
arch=x86_64
arch_build=x86_64
compiler=gcc
compiler.version=4.9
compiler.exception=seh
compiler.libcxx=libstdc++11
compiler.threads=posix
build_type=Release
```

Then you can have a *conanfile.py* that can use `self.run()` with `win_bash=True` to run any command in a bash terminal or use the `AutoToolsBuildEnvironment` to invoke `configure/make` in the `subsystem`:

```
from conans import ConanFile
import os


class MyToolchainXXXConan(ConanFile):
    name = "mylib"
    version = "0.1"
    ...

    def build(self):
        self.run("some_command", win_bash=True)

        env_build = AutoToolsBuildEnvironment(self, win_bash=True)
        env_build.configure()
        env_build.make()

     ...
```

And apply the profile in your recipe to create a package using the MSYS2 and MINGW:

```
$ conan create . user/testing --profile msys2_mingw
```

As we included in the profile the `MinGW` and then the `MSYS2` build_require, when we run a command, the PATH will contain first the MinGW tools and finally the MSYS2.

What could we do with the Visual Studio issue with `link.exe`? You can pass an additional parameter to `run_in_windows_bash` with a dictionary of environment variables to have more priority than the others:

```
def build(self):
    # ...
    vs_path = tools.vcvars_dict(self.settings)["PATH"] # Extract the path from the␣
↪vcvars_dict tool
    tools.run_in_windows_bash(self, command, env={"PATH": vs_path})
```

So you will get first the `link.exe` from the Visual Studio.

Also, Conan has a tool `tools.remove_from_path` that you can use in a recipe to remove temporally a tool from the path if you know that it can interfere with your build script:

```
class MyToolchainXXXConan(ConanFile):
    name = "mylib"
    version = "0.1"
    ...

    def build(self):
        with tools.remove_from_path("link"):
            # Call something
            self.run("some_command", win_bash=True)

     ...
```

CHAPTER

ELEVEN

INTEGRATIONS

11.1  CMake

Conan can be integrated with CMake using generators, build helpers and custom *findXXX.cmake* files:

11.1.1 `cmake` generator

If you are using **CMake** to build your project, you can use the `cmake` generator to define all your requirements information in cmake syntax. It creates a file named `conanbuildinfo.cmake` that can be imported from your `CMakeLists.txt`.

conanfile.txt

```
...

[generators]
cmake
```

When **conan install** is executed, a file named `conanbuildinfo.cmake` is created.

We can include `conanbuildinfo.cmake` in our project's `CMakeLists.txt` to manage our requirements. The inclusion of `conanbuildinfo.cmake` doesn't alter cmake environment at all, it just provides `CONAN_` variables and some useful macros.

Global variables approach

The simplest way to consume it would be to invoke the `conan_basic_setup()` macro, which will basically set global include directories, libraries directories, definitions, etc. so typically is enough to do:

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

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

The `conan_basic_setup()` is split in smaller macros, that should be self explanatory. If you need to do something different, you can just use them individually.

Targets approach

For **modern cmake (>=3.1.2)**, you can use the following approach:

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

add_executable(timer timer.cpp)
target_link_libraries(timer CONAN_PKG::Poco)
```

Using `TARGETS` as argument, `conan_basic_setup()` will internally call the macro `conan_define_targets()` which defines cmake `INTERFACE IMPORTED` targets, one per package. These targets, named `CONAN_PKG::PackageName` can be used to link with, instead of using global cmake setup.

See also:

Check the section *Reference/Generators/cmake* to read more about this generator.

11.1.2 `cmake_multi` generator

`cmake_multi` generator is intended for CMake multi-configuration environments, like Visual Studio and XCode IDEs that do not configure for a specific `build_type`, like Debug or Release, but rather can be used for both and switch among Debug andRelease configurations with a combo box or similar control. The project configuration for cmake is different, in multi-configuration environments, the flow would be:

```
$ cmake .. -G "Visual Studio 14 Win64"
# Now open the IDE (.sln file) or
$ cmake --build . --config Release
```

While in single-configuration environments (Unix Makefiles, etc):

```
$ cmake .. -G "Unix Makefiles" -DCMAKE_BUILD_TYPE=Release
# Build from your IDE, launching make, or
$ cmake --build .
```

The `CMAKE_BUILD_TYPE` default, if not specified is `Debug`.

With the regular conan `cmake` generator, only 1 configuration at a time can be managed. Then, it is a universal, homogeneous solution for all environments. This is the recommended way, using the regular `cmake` generator, and just go to the command line and switch among configurations:

```
$ conan install -s build_type=Release ...
# Work in release, then, to switch to Debug dependencies
$ conan install -s build_type=Debug ...
```

However, end consumers with heavy usage of the IDE, might want a multi-configuration build. The `cmake_multi` **experimental** generator is able to do that. First, both Debug and Release dependencies have to be installed:

```
$ conan install -g cmake_multi -s build_type=Release ...
$ conan install -g cmake_multi -s build_type=Debug  ...
```

These commands will generate 3 files: `conanbuildinfo_release.cmake`, `conanbuildinfo_debug.cmake`, and `conanbuildinfo_multi.cmake`, which includes the other two, and enables its use.

> **Warning:** The `cmake_multi` generator is designed as a helper for consumers, but not for creating packages. If you also want to create a package, see *Creating packages* section.

Global variables approach

The consumer project might write a `CMakeLists.txt` like:

```
project(MyHello)
cmake_minimum_required(VERSION 2.8.12)

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

add_executable(say_hello main.cpp)
foreach(_LIB ${CONAN_LIBS_RELEASE})
    target_link_libraries(say_hello optimized ${_LIB})
endforeach()
foreach(_LIB ${CONAN_LIBS_DEBUG})
    target_link_libraries(say_hello debug ${_LIB})
endforeach()
```

Targets approach

Or, if using the modern cmake syntax with targets (where `Hello1` is an example package name that the executable `say_hello` depends on):

```
project(MyHello)
cmake_minimum_required(VERSION 2.8.12)

include(${CMAKE_BINARY_DIR}/conanbuildinfo_multi.cmake)
conan_basic_setup(TARGETS)

add_executable(say_hello main.cpp)
target_link_libraries(say_hello CONAN_PKG::Hello1)
```

There's also a convenient macro for linking to all libraries:

```
project(MyHello)
cmake_minimum_required(VERSION 2.8.12)

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

add_executable(say_hello main.cpp)
conan_target_link_libraries(say_hello)
```

With this approach, the end user can open the generated IDE project and switch among both configurations, building the project, or from the command line:

```
$ cmake --build . --config Release
# And without having to conan install again, or do anything else
$ cmake --build . --config Debug
```

Creating packages

The `cmake_multi` generator is just for consumption. It cannot be used to create packages. If you want to be able to both use the `cmake_multi` generator to install dependencies and build your project but also to create packages

from that code, you need to specify the regular `cmake` generator for package creation, and prepare the *CMakeLists.txt* accordingly, something like:

```
project(MyHello)
cmake_minimum_required(VERSION 2.8.12)

if(EXISTS ${CMAKE_BINARY_DIR}/conanbuildinfo_multi.cmake)
    include(${CMAKE_BINARY_DIR}/conanbuildinfo_multi.cmake)
else()
    include(${CMAKE_BINARY_DIR}/conanbuildinfo.cmake)
endif()

conan_basic_setup()

add_executable(say_hello main.cpp)
conan_target_link_libraries(say_hello)
```

Then, make sure that the generator `cmake_multi` is **not** specified in the conanfiles, but the users specify it in the command line while installing dependencies:

```
$ conan install . -g cmake_multi
```

See also:

Check the section *Reference/Generators/cmake* to read more about this generator.

11.1.3 Build automation

You can invoke CMake from your conanfile.py file and automate the build of your library/project. Conan provides a `CMake()` helper. This helper is useful to call `cmake` command both for creating conan packages or automating your project build with the **`conan build .`** command. The `CMake()` helper will take into account your settings to automatically set definitions and a generator according to your compiler, build_type, etc.

See also:

Check the section *Building with CMake*.

11.1.4 Find Packages

If a FindXXX.cmake file for the library you are packaging is already available, it should work automatically.

Variables **CMAKE_INCLUDE_PATH** and **CMAKE_LIBRARY_PATH** are set with the right requirements paths. CMake **find_library** function will be able to locate the libraries in the package's folders.

So, you can use **find_package** normally:

```
project(MyHello)
cmake_minimum_required(VERSION 2.8.12)

include(conanbuildinfo.cmake)
conan_basic_setup()

find_package("ZLIB")

if(ZLIB_FOUND)
    add_executable(enough enough.c)
    include_directories(${ZLIB_INCLUDE_DIRS})
```

(continues on next page)

(continued from previous page)

```
    target_link_libraries(enough ${ZLIB_LIBRARIES})
else()
    message(FATAL_ERROR "Zlib not found")
endif()
```

In addition to automatic **find_package** support, **CMAKE_MODULE_PATH** variable is set with your requirements root package paths. You can override the default behavior of any find_package() by creating a `findXXX.cmake` file in your package.

Creating a custom FindXXX.cmake file

Sometimes the "official" CMake FindXXX.cmake scripts are not ready to find our libraries (not supported library names for specific settings, fixed installation directories like C:\OpenSSL... etc) Or maybe there is no "official" CMake script for our library.

So in these cases we can provide a custom **FindXXX.cmake** file in our conan packages.

1. Create a file named FindXXX.cmake and save it in your conan package root folder. Where XXX is the name of the library that we will use in the **find_package** CMake function. For example, we create a `FindZLIB.cmake` and use `find_package(ZLIB)`. We recommend to copy the original FindXXX.cmake file from Kitware (folder Modules/FindXXX.cmake), if available, and modify it to help finding our library files, but it depends a lot, maybe you are interested in creating a new one.

If it's not provided you can create a basic one, take a look at this example with the ZLIB library:

FindZLIB.cmake

```
find_path(ZLIB_INCLUDE_DIR NAMES zlib.h PATHS ${CONAN_INCLUDE_DIRS_ZLIB})
find_library(ZLIB_LIBRARY NAMES ${CONAN_LIBS_ZLIB} PATHS ${CONAN_LIB_DIRS_ZLIB})

set(ZLIB_FOUND TRUE)
set(ZLIB_INCLUDE_DIRS ${ZLIB_INCLUDE_DIR})
set(ZLIB_LIBRARIES ${ZLIB_LIBRARY})
mark_as_advanced(ZLIB_LIBRARY ZLIB_INCLUDE_DIR)
```

In the first line we are finding the path where our headers should be found, we suggest the CONAN_INCLUDE_DIRS_XXX. Then the same for the library names with CONAN_LIBS_XXX and the paths where the libs are CONAN_LIB_DIRS_XXX.

2. In your conanfile.py file add the `FindXXX.cmake` to the `exports_sources` field:

```
class HelloConan(ConanFile):
    name = "Hello"
    version = "0.1"
    ...
    exports_sources = ["FindXXX.cmake"]
```

3. In the package method, copy the `FindXXX.cmake` file to the root:

```
class HelloConan(ConanFile):
    name = "Hello"
    version = "0.1"
    ...
    exports_sources = ["FindXXX.cmake"]
```

(continues on next page)

(continued from previous page)

```
    def package(self):
        ...
        self.copy("FindXXX.cmake", ".", ".")
```

11.2 Autotools: configure/make

If you are using **configure/make** you can use **AutoToolsBuildEnvironment** helper. This helper sets `LIBS`, `LDFLAGS`, `CFLAGS`, `CXXFLAGS` and `CPPFLAGS` environment variables based on your requirements.

Check *Building with Autotools* for more info.

11.3 Visual Studio

Conan can be integrated with **Visual Studio** in two different ways:

- Using the **cmake** generator to create a **conanbuildinfo.cmake** file.
- Using the **visual_studio** generator to create a **conanbuildinfo.props** file.

11.3.1 With CMake

Use the **cmake** generator, or **cmake_multi**, if you are using cmake to machine-generate your Visual Studio projects.

Check the *generator* section to read about the **cmake** generator. Check the official CMake docs to find out more about generating Visual Studio projects with CMake.

However, beware of some current cmake limitations, such as not dealing well with find-packages, because cmake doesn't know how to handle finding both debug and release packages.

Note: If you want to use the Visual Studio 2017 + CMake integration, *check this how-to*

11.3.2 With *visual_studio* generator

Use this, or **visual_studio_multi**, if you are maintaining your Visual Studio projects, and want to use Conan to to tell Visual Studio how to find your third-party dependencies.

You can use the **visual_studio** generator to manage your requirements via your *Visual Studio* project.

This generator creates a Visual Studio project properties file, with all the *include paths*, *lib paths*, *libs*, *flags* etc, that can be imported in your project.

Open `conanfile.txt` and change (or add) the **visual_studio** generator:

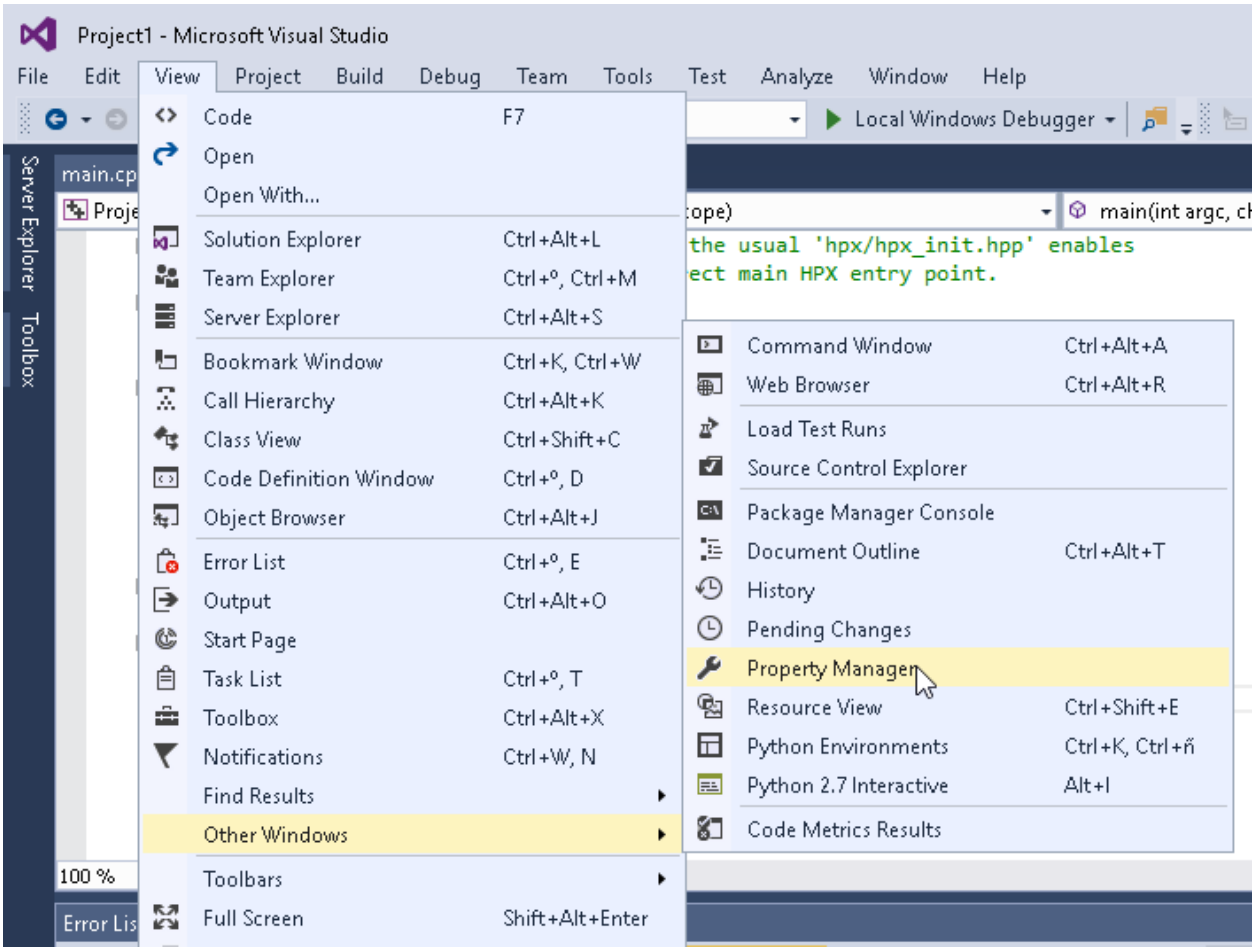
```
[requires]
Poco/1.7.8p3@pocoproject/stable

[generators]
visual_studio
```

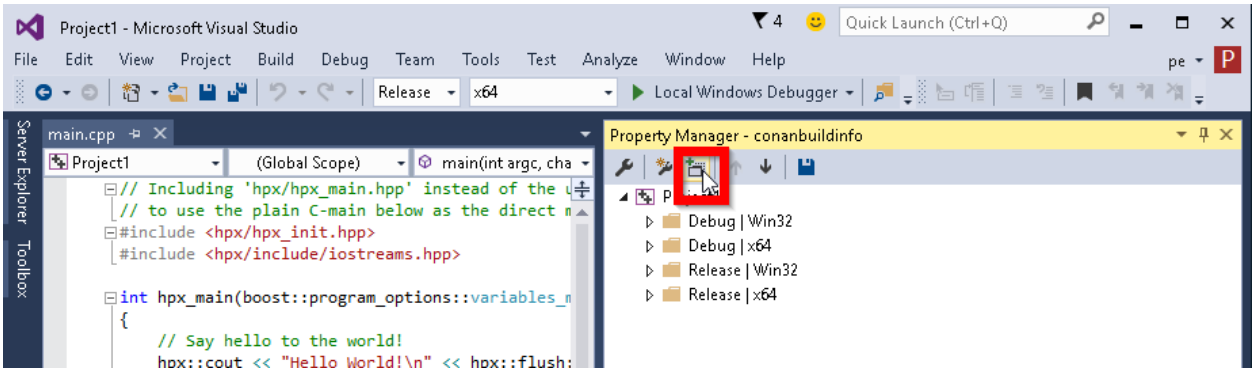
Install the requirements:

```
$ conan install
```

Go to your Visual Studio project, and open the **Property Manager**, usually in **View -> Other Windows -> Property Manager**.



Click the **"+"** icon and select the generated `conanbuildinfo.props` file:



Build your project as usual.

Note: Remember to set your project's architecture and build type accordingly, explicitly or implicitly, when issuing the **conan install** command. If these values don't match, you build will probably fail.

e.g. **Release/x64**

See also:

Check the *Reference/Generators/visual_studio* for the complete reference.

11.3.3 Calling Visual Studio compiler

You can call your Visual Studio compiler from your `build()` method using the `VisualStudioBuildEnvironment` and the `tools.vcvars_command`.

Check *Build with Visual Studio* section for more info.

11.3.4 Build an existing Visual Studio project

You can build an existing Visual Studio from your `build()` method using the *MSBuild()* build helper.

```
from conans import ConanFile, MSBuild

class ExampleConan(ConanFile):
    ...

    def build(self):
        msbuild = MSBuild(self)
        msbuild.build("MyProject.sln")
```

11.3.5 Toolsets

You can use the subsetting `toolset` of the Visual Studio compiler to specify a custom toolset. It will be automatically applied when using the `CMake()` and `MSBuild()` build helpers. The toolset can be also specified manually in these build helpers with the `toolset` parameter.

By default, Conan will not generate a new binary package if the specified `compiler.toolset` matches an already generated package for the corresponding `compiler.version`. Check the *package_id()* reference to know more.

See also:

- Check the *CMake()* reference section for more info.

11.4 Xcode

Conan can be integrated with **XCode** in two different ways:

- Using the **cmake** generator to create a **conanbuildinfo.cmake** file.
- Using the **xcode** generator to create a **conanbuildinfo.xcconfig** file.

11.4.1 With CMake

Check the *Integrations/cmake* section to read about the **cmake** generator. Check the official CMake docs to find out more about generating Xcode projects with CMake.

11.4.2 With the *xcode* generator

You can use the **xcode** generator to integrate your requirements in your *Xcode* project. This generator creates an `xcconfig` file, with all the *include paths*, *lib paths*, *libs*, *flags* etc, that can be imported in your project.

Open `conanfile.txt` and change (or add) the **xcode** generator:

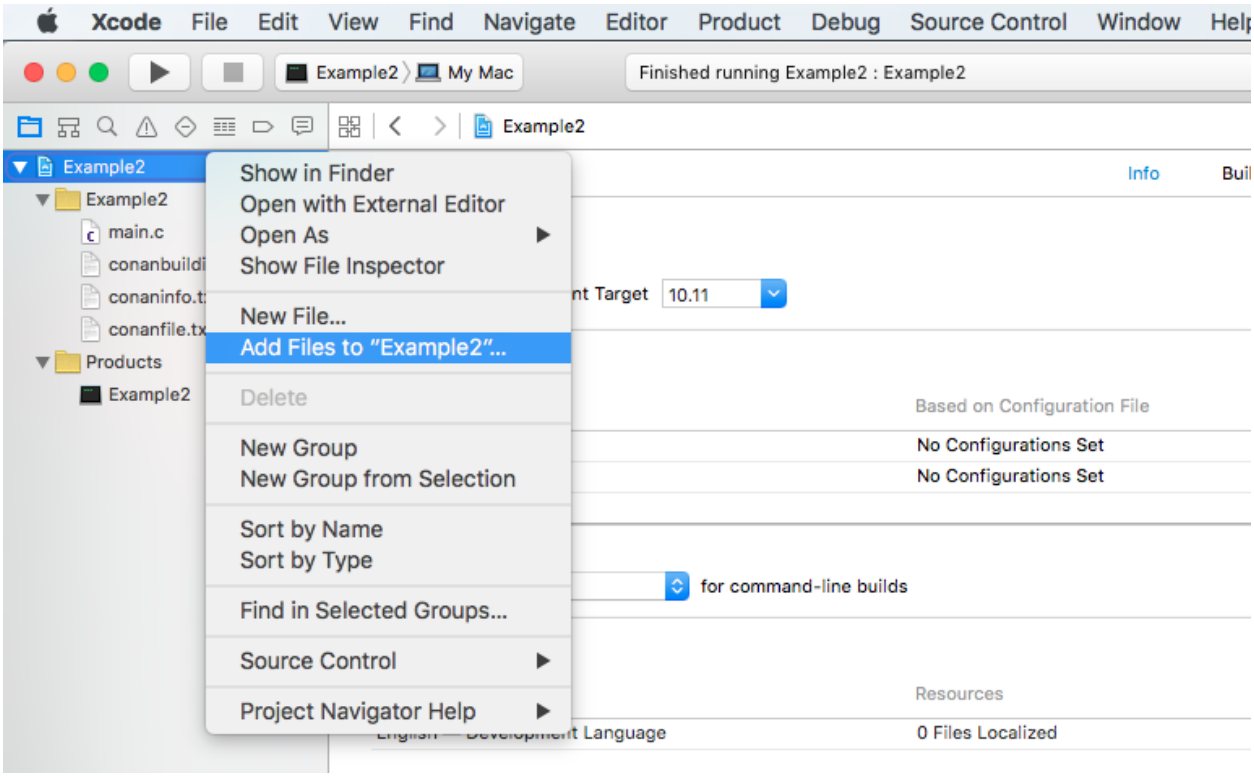
```
[requires]
Poco/1.7.8p3@pocoproject/stable

[generators]
xcode
```

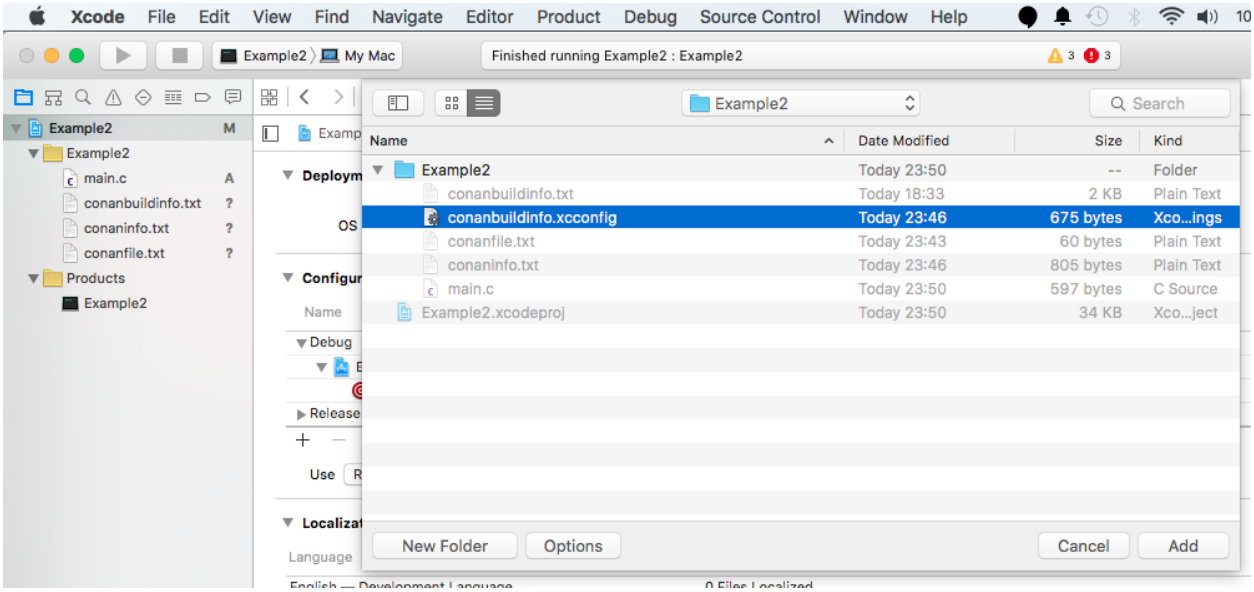
Install the requirements:

```
$ conan install
```

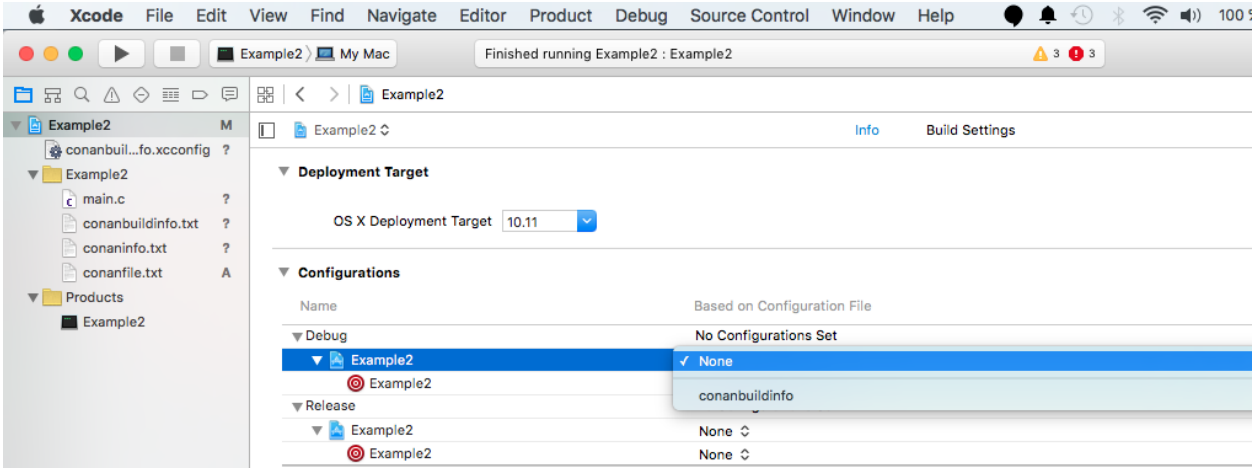
Go to your **Xcode** project, click on the project and select **Add files to**.



Choose `conanbuildinfo.xcconfig` generated.



Click on the project again. In the **info/configurations** section, choose **conanbuildinfo** for *release* and *debug*.



Build your project as usual.

See also:

Check the *Reference/Generators/xcode* for the complete reference.

11.5 Compilers on command line

The **compiler_args** generator creates a file named `conanbuildinfo.args` containing a command line arguments to invoke `gcc`, `clang` or `cl` (Visual Studio) compiler.

Now we are going to compile the *getting started* example using **compiler_args** instead of the **cmake** generator.

Open `conanfile.txt` and change (or add) **compiler_args** generator:

```
[requires]
Poco/1.9.0@pocoproject/stable

[generators]
compiler_args
```

Install the requirements (from the mytimer/build folder):

```
$ conan install ..
```

Note: Remember, if you don't specify settings in **install command** with **-s**, conan will use the detected defaults. You can always change them by editing the `~/.conan/profiles/default` or override them with "-s" parameters.

The generated `conanbuildinfo.args`:

```
-DPOCO_STATIC=ON -DPOCO_NO_AUTOMATIC_LIBS
-Ipath/to/Poco/1.7.9/pocoproject/stable/package/
↪dd758cf2da203f96c86eb99047ac152bcd0c0fa9/include
-Ipath/to/OpenSSL/1.0.2l/conan/stable/package/
↪227fb0ea22f4797212e72ba94ea89c7b3fbc2a0c/include
-Ipath/to/zlib/1.2.11/conan/stable/package/8018a4df6e7d2b4630a814fa40c81b85b9182d2b/
↪include
-m64 -DNDEBUG -Wl,-rpath,"path/to/Poco/1.7.9/pocoproject/stable/package/
↪dd758cf2da203f96c86eb99047ac152bcd0c0fa9/lib"
```

(continues on next page)

(continued from previous page)

```
-Wl,-rpath,"path/to/OpenSSL/1.0.2l/conan/stable/package/
↪227fb0ea22f4797212e72ba94ea89c7b3fbc2a0c/lib"
-Wl,-rpath,"path/to/zlib/1.2.11/conan/stable/package/
↪8018a4df6e7d2b4630a814fa40c81b85b9182d2b/lib"
-Lpath/to/Poco/1.7.9/pocoproject/stable/package/
↪dd758cf2da203f96c86eb99047ac152bcd0c0fa9/lib
-Lpath/to/OpenSSL/1.0.2l/conan/stable/package/
↪227fb0ea22f4797212e72ba94ea89c7b3fbc2a0c/lib
-Lpath/to/zlib/1.2.11/conan/stable/package/8018a4df6e7d2b4630a814fa40c81b85b9182d2b/
↪lib
-lPocoUtil -lPocoMongoDB -lPocoNet -lPocoNetSSL -lPocoCrypto -lPocoData -
↪lPocoDataSQLite -lPocoZip
-lPocoXML -lPocoJSON -lPocoFoundation -lssl -lcrypto -lz -stdlib=libc++
```

This is hard to read, but those are just the **compiler_args** parameters needed to compile our program:

- **-I** options with headers directories
- **-L** for libraries directories
- **-l** for library names
- and so on... see the *complete reference here*

It's almost the same information we can see in `conanbuildinfo.cmake` and many other generators' files.

Run:

```
$ mkdir bin
$ g++ ../timer.cpp @conanbuildinfo.args -std=c++14 -o bin/timer
```

Note: "@conanbuildinfo.args" appends all the file contents to g++ command parameters

```
$ ./bin/timer
 Callback called after 250 milliseconds.
 ...
```

To invoke `cl` (Visual Studio compiler):

```
$ cl /EHsc timer.cpp @conanbuildinfo.args
```

You can also use the generator within your `build()` method of your conanfile.py.

Check the *Reference, generators, compiler_args* section for more info.

11.6 Android Studio

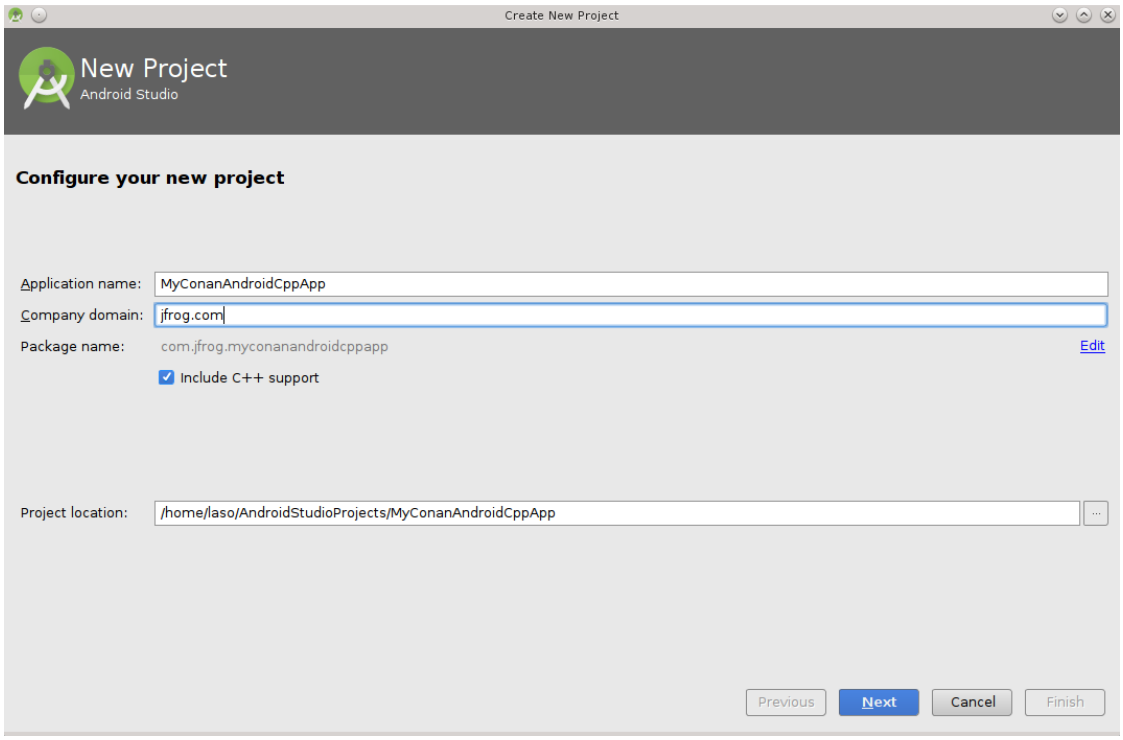
You can use Conan to *cross-build your libraries for Android* in different architectures. If you are using Android Studio for your Android application development, you can integrate it conan to automate the library building for the different

architectures that you want to support in your project.

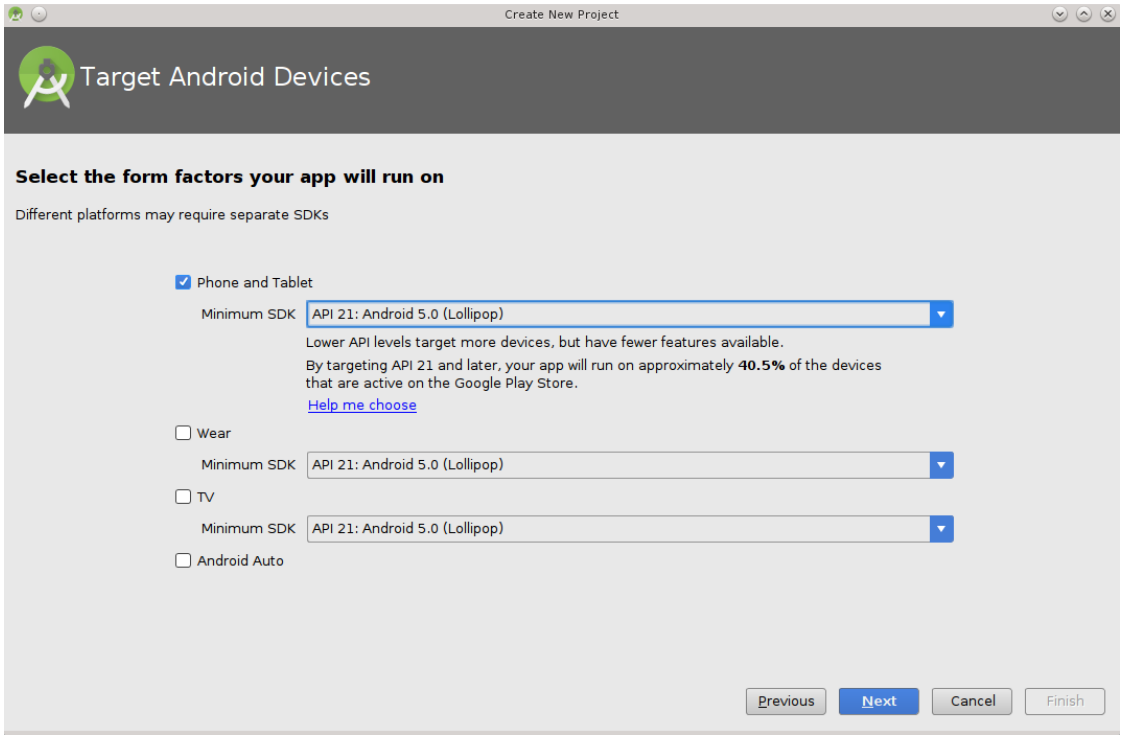
Here is an example of how to integrate the `libpng` conan package library in an Android application, but any library that can be cross-compiled to Android could be used using the same procedure.

We are going to start from the "Hello World" wizard application and then will add it the `libpng` C library:

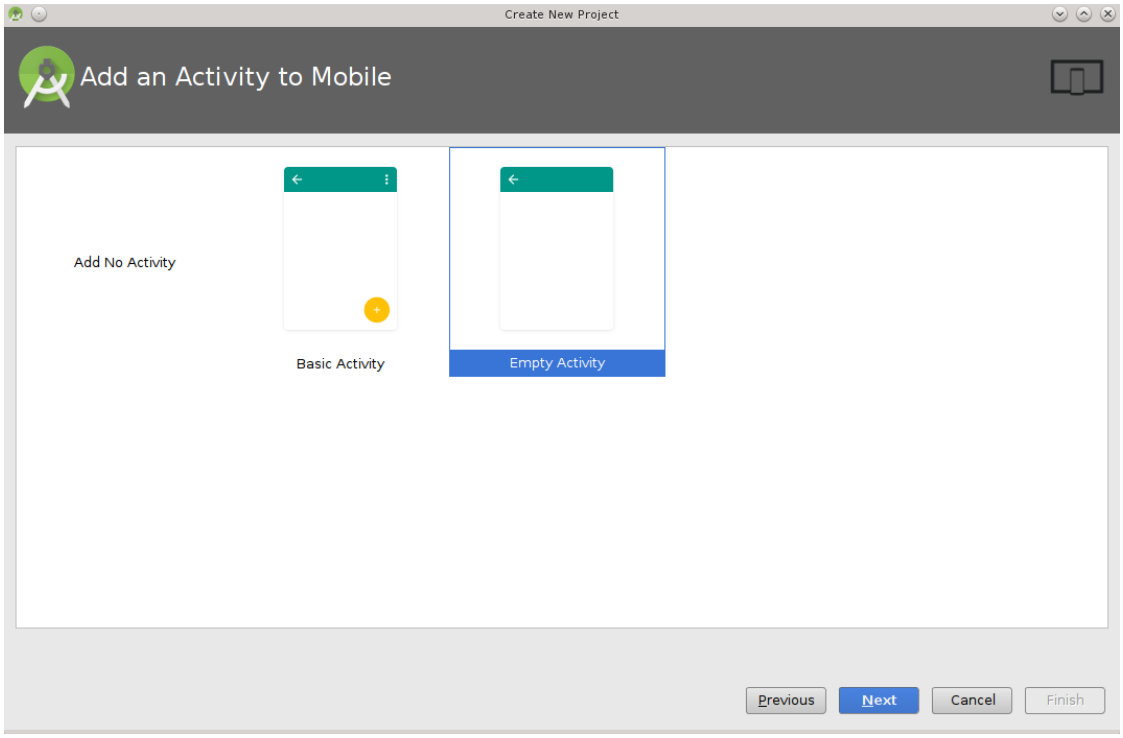
1. Follow the *cross-build your libraries for Android* guide to create a standalone toolchain and create a profile `android_21_arm_clang` for Android. You can also use the NDK that the Android Studio installs.
2. Create a new Android Studio project and include C++ support.

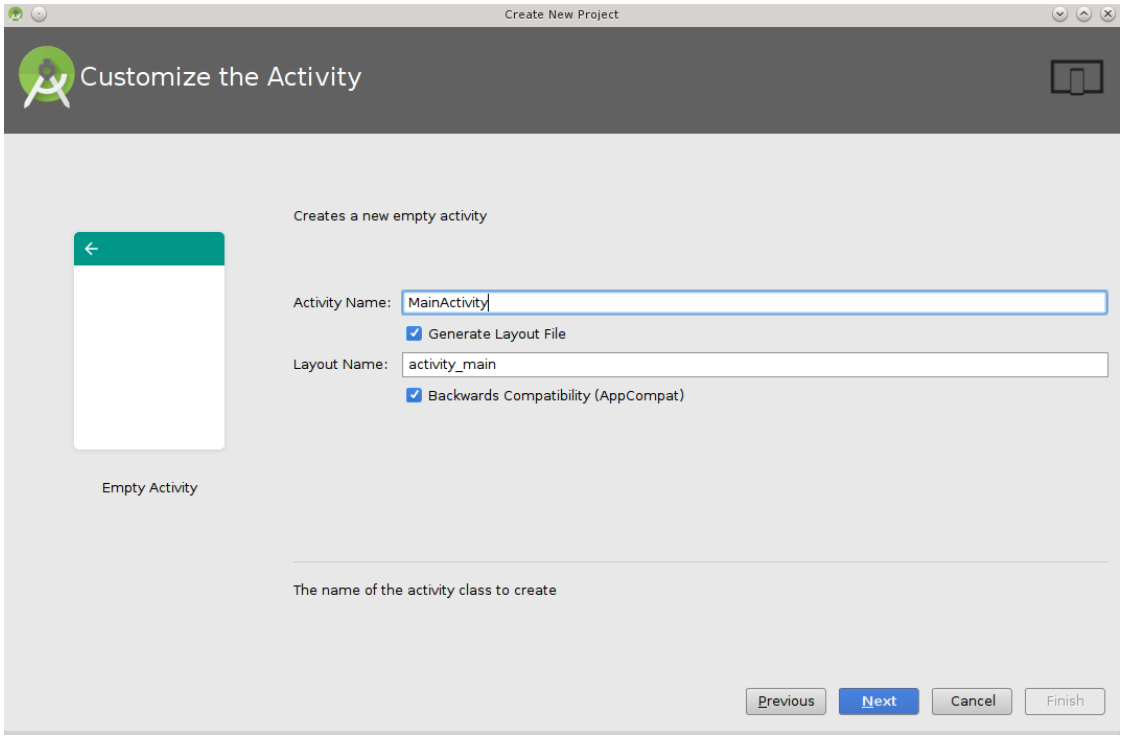


3. Select your API level and target, the arch and api level have to match with the standalone toolchain created in step 1.

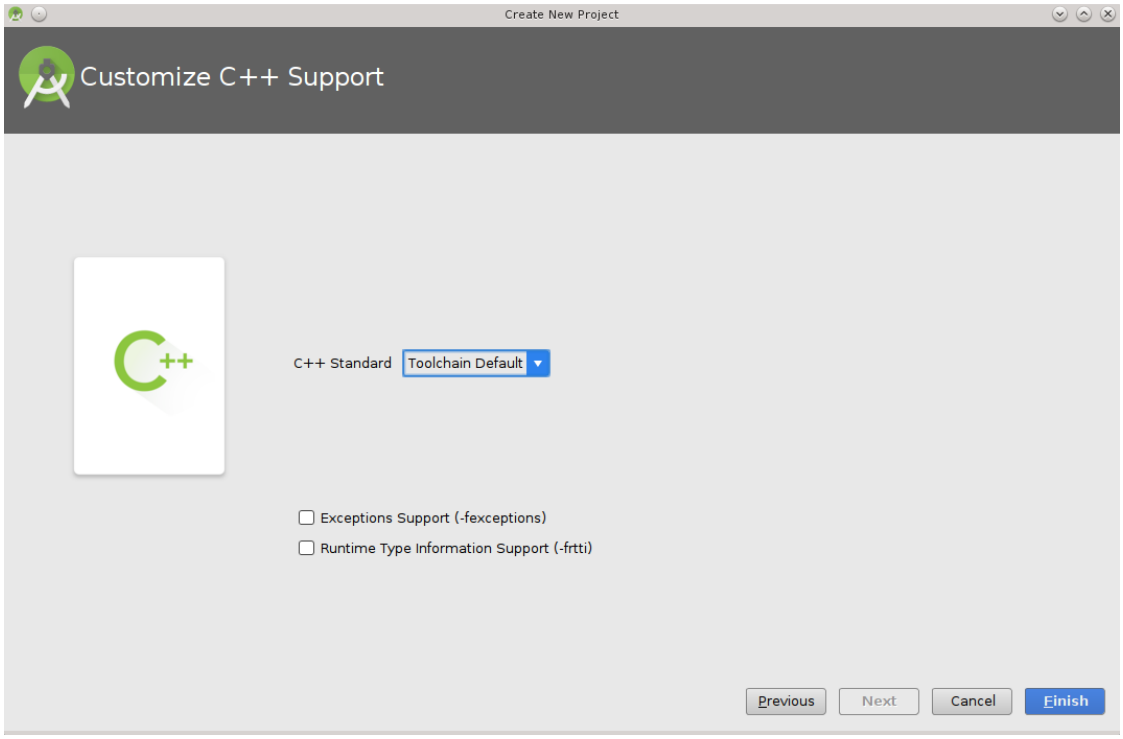


4. Add an empty Activity and name it.





5. Select the C++ standard



6. Change to the *project view* and in the *app* folder create a `conanfile.txt` with the following contents:

conanfile.txt

```
[requires]
```

(continues on next page)

(continued from previous page)

```
libpng/1.6.23@lasote/stable

[generators]
cmake
```

7. Open the `CMakeLists.txt` file from the app folder and replace the contents with:

```
cmake_minimum_required(VERSION 3.4.1)

include(${CMAKE_CURRENT_SOURCE_DIR}/conan_build/conanbuildinfo.cmake)
set(CMAKE_CXX_COMPILER_VERSION "5.0") # Unknown miss-detection of the compiler by␣
↪CMake
conan_basic_setup(TARGETS)

add_library(native-lib SHARED src/main/cpp/native-lib.cpp)
target_link_libraries(native-lib CONAN_PKG::libpng)
```

8. Open the *app/build.gradle* file, we are configuring the architectures we want to build specifying adding a new task `conanInstall` that will call **`conan install`** to install the requirements:

- In the defaultConfig section, append:

```
ndk {
   // Specifies the ABI configurations of your native
   // libraries Gradle should build and package with your APK.
   abiFilters 'armeabi-v7a'
}
```

- After the android block:

```
task conanInstall {
    def buildDir = new File("app/conan_build")
    buildDir.mkdirs()
    // if you have problems running the command try to specify the absolute
    // path to conan (Known problem in MacOSX) /usr/local/bin/conan
    def cmmd = "conan install ../conanfile.txt --profile android_21_arm_clang --build␣
↪missing "
    print(">> ${cmmd} \n")

    def sout = new StringBuilder(), serr = new StringBuilder()
    def proc = cmmd.execute(null, buildDir)
    proc.consumeProcessOutput(sout, serr)
    proc.waitFor()
    println "$sout $serr"
    if(proc.exitValue() != 0){
        throw new Exception("out> $sout err> $serr" + "\nCommand: ${cmmd}")
    }
}
```

9. Finally open the default example cpp library in `app/src/main/cpp/native-lib.cpp` and include some lines using your library. Be careful with the JNICALL name if you used other app name in the wizard:

```
#include <jni.h>
#include <string>
#include "png.h"
#include "zlib.h"
#include <sstream>
```

(continues on next page)

(continued from previous page)

```
#include <iostream>

extern "C"
JNIEXPORT jstring JNICALL
Java_com_jfrog_myconanandroidcppapp_MainActivity_stringFromJNI(
        JNIEnv *env,
        jobject /* this */) {
    std::ostringstream oss;
    oss << "Compiled with libpng: " << PNG_LIBPNG_VER_STRING << std::endl;
    oss << "Running with libpng: " << png_libpng_ver << std::endl;
    oss << "Compiled with zlib: " << ZLIB_VERSION << std::endl;
    oss << "Running with zlib: " << zlib_version << std::endl;

    return env->NewStringUTF(oss.str().c_str());
}
```

Build your project normally, conan will create a "conan" folder with a folder for each different architecture you have speified in the abiFilters with a `conanbuildinfo.cmake` file.

Then run the app using an x86 emulator for best performance:

See also:

Check the section *howtos/Cross building/Android* to read more about cross building for Android.

11.7 CLion

CLion uses **CMake** as the build system of projects, so you can use the *CMake generator* to manage your requirements in your CLion project.

Just include the `conanbuildinfo.cmake` this way:

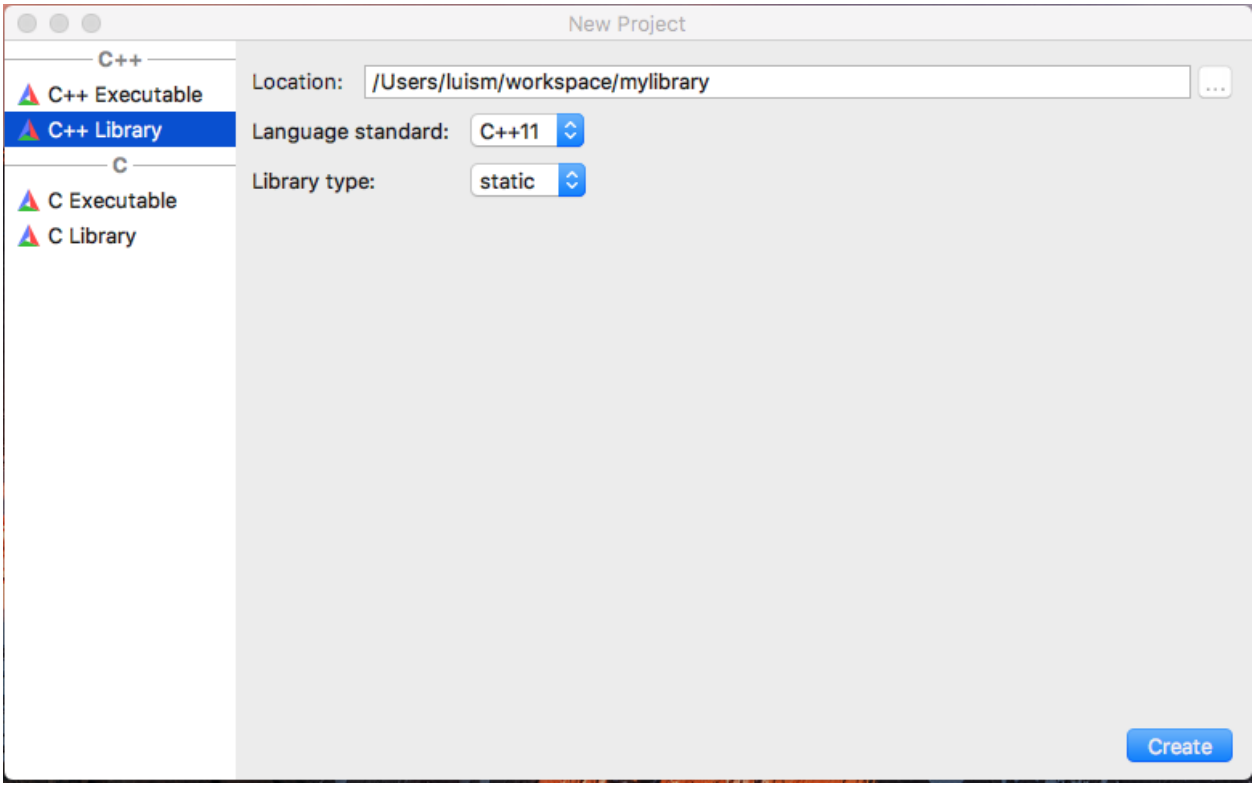
```
if(EXISTS ${CMAKE_BINARY_DIR}/conanbuildinfo.cmake)
    include(${CMAKE_BINARY_DIR}/conanbuildinfo.cmake)
    conan_basic_setup()
else()
    message(WARNING "The file conanbuildinfo.cmake doesn't exist, you have to run␣
↪conan install first")
endif()
```

If the `conanbuildinfo.cmake` file is not found, it will print a warning message in the `Messages` console of your Clion IDE.

11.7.1 Using packages in a CLion project

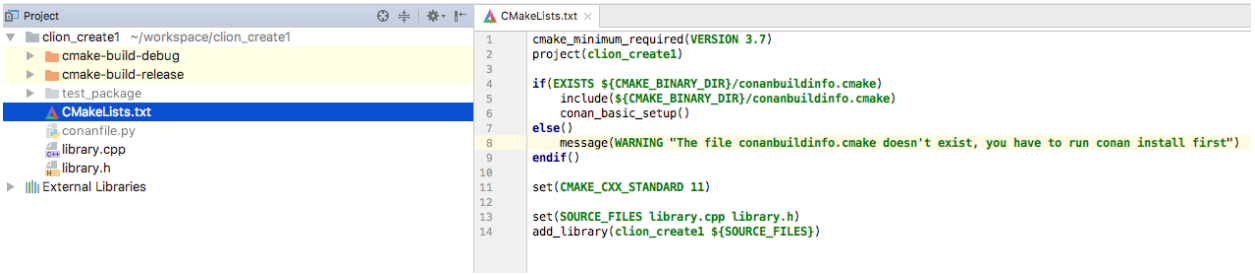
Let see an example of how to consume Conan packages in a CLion project. We are going to require and use the `zlib` conan package.

1. Create a new CLion project

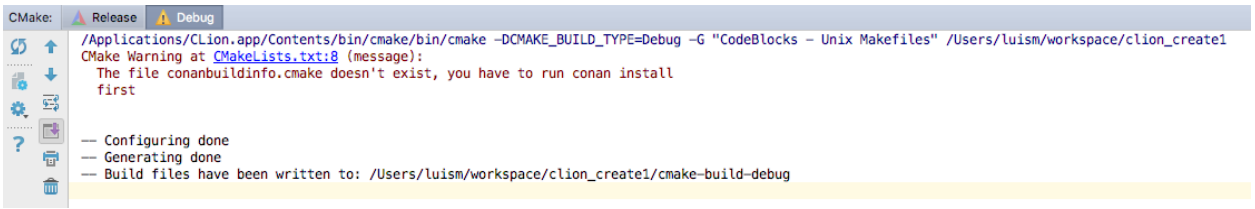


2. Edit the `CMakeLists.txt` file and add the following lines:

```
if(EXISTS ${CMAKE_BINARY_DIR}/conanbuildinfo.cmake)
    include(${CMAKE_BINARY_DIR}/conanbuildinfo.cmake)
    conan_basic_setup()
else()
    message(WARNING "The file conanbuildinfo.cmake doesn't exist, you have to run␣
↪conan install first")
endif()
```



3. CLion will reload your CMake project and you will be able to see a Warning in the console, because the `conanbuildinfo.cmake` file still doesn't exists:

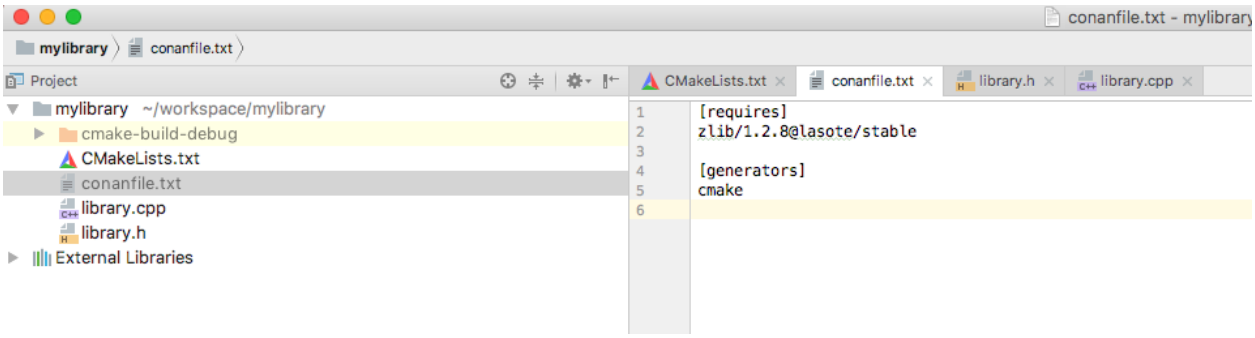


4. Create a `conanfile.txt` with all your requirements and use the `cmake` generator. In this case we are only

requiring zlib library from a conan package:

```
[requires]
zlib/1.2.11@conan/stable

[generators]
cmake
```



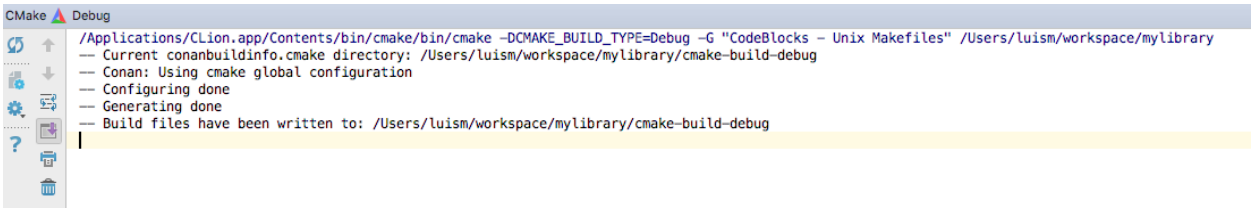
5. Now you can **`conan install`** for debug in the `cmake-build-debug` folder to install your requirements and generate the `conanbuildinfo.cmake` file there:

```
$ conan install . -s build_type=Debug --install-folder=cmake-build-debug
```

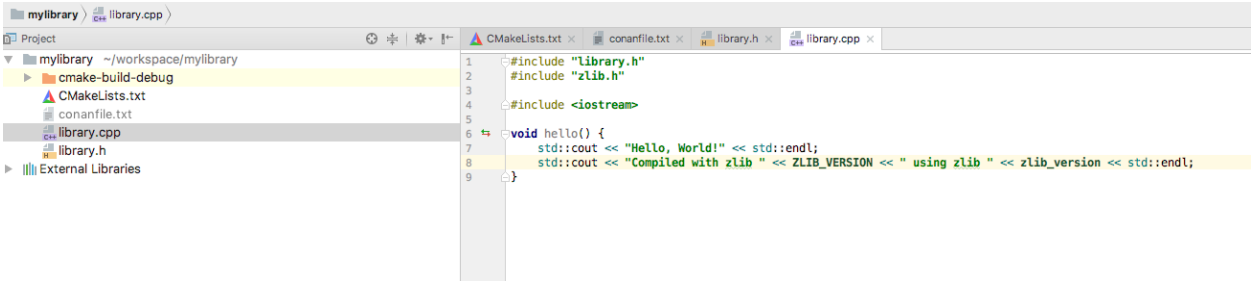
6. Repeat the last step if you have the release build types configured in your CLion IDE, but changing the build_type setting accordingly:

```
$ conan install . -s build_type=Release --install-folder=cmake-build-release
```

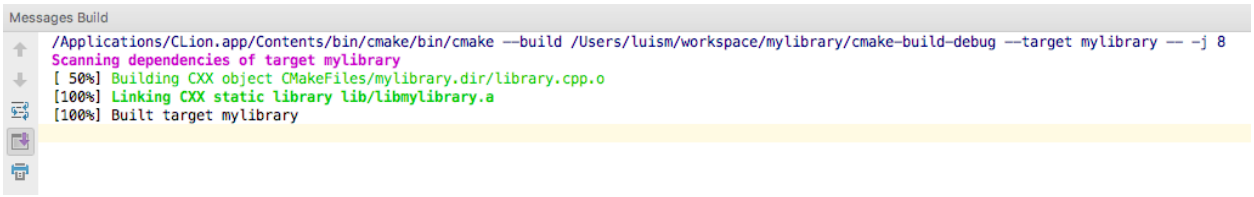
7. Now reconfigure your CLion project, the Warning message is not shown anymore:



8. Open the `library.cpp` file and include the `zlib.h`, if you follow the link you can see that CLion automatically detect the `zlib.h` header file from the local conan cache.



9. Build your project normally using your CLion IDE:

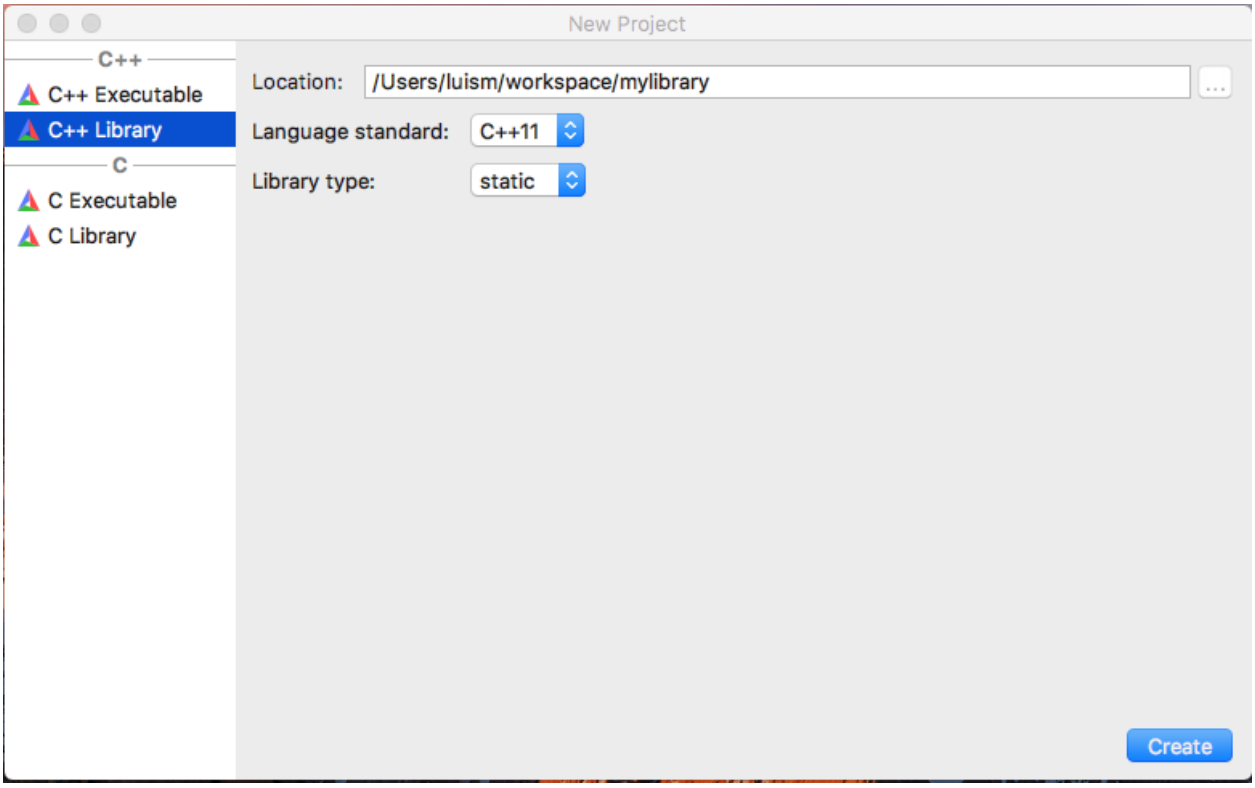


You can check a full example of a CLion project reusing conan packages in this github repository: lasote/clion-conan-consumer.

11.7.2 Creating conan packages in a CLion project

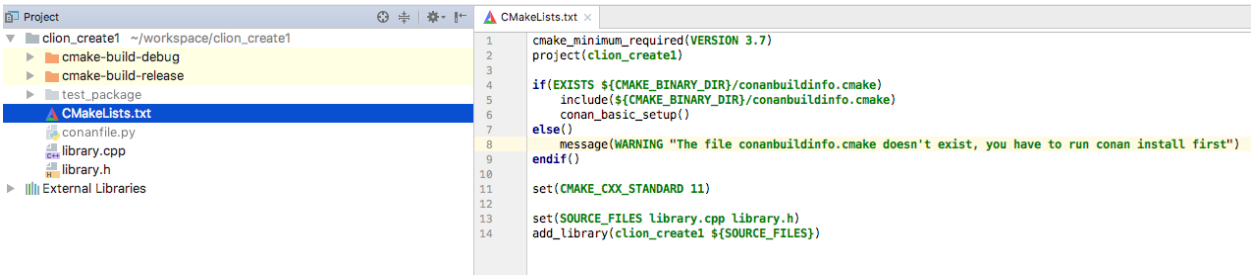
Now we are going to see how to create a conan package from the previous library.

1. Create a new CLion project



2. Edit the `CMakeLists.txt` file and add the following lines:

```
if(EXISTS ${CMAKE_BINARY_DIR}/conanbuildinfo.cmake)
    include(${CMAKE_BINARY_DIR}/conanbuildinfo.cmake)
    conan_basic_setup()
else()
    message(WARNING "The file conanbuildinfo.cmake doesn't exist, you have to run␣
↪conan install first")
endif()
```



3. Create a `conanfile.py` file. It's recommended to use the **`conan new`** command.

```
$ conan new mylibrary/1.0@myuser/channel
```

And edit the `conanfile.py`:

- We are removing the `source` method because we have the sources in the same project, so we can use the `exports_sources`.
- In the `package_info` method adjust the library name, in this case our `CMakeLists.txt` is creating a target library called `mylibrary`.
- Adjust the CMake helper in the `build()` method, the `cmake.configure()` doesn't need to specify the `source_folder`, because we have the `library.*` files in the root directory.
- Adjust the `copy` function calls in the `package` method to ensure that all your headers and libraries are copied to the conan package.

```
from conans import ConanFile, CMake, tools


class MylibraryConan(ConanFile):
    name = "mylibrary"
    version = "1.0"
    license = "<Put the package license here>"
    url = "<Package recipe repository url here, for issues about the package>"
    description = "<Description of Mylibrary here>"
    settings = "os", "compiler", "build_type", "arch"
    options = {"shared": [True, False]}
    default_options = "shared=False"
    generators = "cmake"
    requires = "zlib/1.2.11@conan/stable"

    def build(self):
        cmake = CMake(self)
        cmake.configure()
        cmake.build()

        # Explicit way:
        # self.run('cmake "%s" %s' % (self.source_folder, cmake.command_line))
        # self.run("cmake --build . %s" % cmake.build_config)

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

(continues on next page)

(continued from previous page)

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

4. To build your library with CLion follow the guide of *Using packages from the step 5*.

5. To package your library use the **conan export-pkg** command passing the used build-folder. It will call your `package()` method to extract the artifacts and push the conan package to the local cache:

```
$ conan export-pkg . mylibrary/1.0@myuser/channel --build-folder cmake-build-debug
```

7. Now you can upload it to a conan server if needed:

```
$ conan upload mylibrary/1.0@myuser/channel # This will upload only the recipe, use --
↪all to upload all the generated binary packages.
```

8. If you would like to see how the package looks like before exporting it to the local cache (conan export-pkg) you can use the **conan package** command to create the package in a local directory:

```
$ conan package . --build-folder cmake-build-debug --package-folder=mypackage
```

If we list the `mypackage` folder we can see:

- A `lib` folder containing our library
- A `include` folder containing our header files
- A `conaninfo.txt` and `conanmanifest.txt` conan files, always present in all packages.

You can check a full example of a CLion project for creating a conan package in this github repository: lasote/clion-conan-package.

11.8 Ninja, NMake, Borland

These build systems still don't have a conan generator for using them natively. However, if you are using cmake, you can instruct conan to use them instead of the default generator (typically `Unix Makefiles`) defining the environment variable `CONAN_CMAKE_GENERATOR`.

Read more about this variable in *Environment variables*.

11.9 pkg-config and pc files

11.9.1 Intro

If you are creating a Conan package for a library (A) and the build system uses `.pc` files to locate its dependencies (B and C), Conan packages too, you can follow different approaches.

The main issue to solve is the absolute paths. When an user installs a package in the local cache, the directory will probably be different from the directory where the package was created, because of the different computer, conan home directory or even different user or channel:

In the machine where the packages were created:

```
/home/user/lasote/.data/storage/zlib/1.2.11/conan/stable
```

In the machine where some user are reusing the library:

```
/custom/dir/.data/storage/zlib/1.2.11/conan/testing
```

So the `.pc` files containing absolute paths won't work to locate the dependencies.

Example of a `.pc` file with an absolute path:

```
prefix=/Users/lasote/.conan/data/zlib/1.2.11/lasote/stable/package/
↪b5d68b3533204ad67e01fa587ad28fb8ce010527
exec_prefix=${prefix}
libdir=${exec_prefix}/lib
sharedlibdir=${libdir}
includedir=${prefix}/include

Name: zlib
Description: zlib compression library
Version: 1.2.11

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

11.9.2 Approach 1: Import and patch the prefix in the *pc* files

Following this approach your library *A* will import to a local directory the `.pc` files from *B* and *C*, then, as they will contain absolute paths, the recipe for *A* will patch the paths to match the current installation directory.

You will need to package the *pc* files from your dependencies. You can adjust the *PKG_CONFIG_PATH* to let `pkg-config` tool locate your `.pc` files.

```
import os
from conans import ConanFile, tools

class LibAConan(ConanFile):
    name = "libA"
    version = "1.0"
    settings = "os", "compiler", "build_type", "arch"
    exports_sources = "*.cpp"
    requires = "libB/1.0@conan/stable"

    def build(self):
        lib_b_path = self.deps_cpp_info["libB"].rootpath
        copyfile(os.path.join(lib_b_path, "libB.pc"), "libB.pc")
        # Patch copied file with the libB path
        tools.replace_prefix_in_pc_file("libB.pc", lib_b_path)

        with tools.environment_append({"PKG_CONFIG_PATH": os.getcwd()}):
            # CALL YOUR BUILD SYSTEM (configure, make etc)
            # E.j: self.run('g++ main.cpp $(pkg-config libB --libs --cflags) -o main')
```

11.9.3 Approach 2: Prepare and package *pc* files before package them

With this approach you will patch the `pc` files from B and C before package them. The goal is to replace the absolute path (the variable part of the path) with a variable placeholder. Then in the consumer package A, declare the variable using `--define-variable` when calling the *pkg-config* command.

This approach is cleaner than approach 1, because the packaged files are already prepared to be reused with or without conan, just declaring the needed variable. And it's not needed to import the `pc` files to the consumer package. However, you need B and C libraries to package the `pc` files correctly.

Library B recipe (preparing the `pc` file):

```
from conans import ConanFile, tools

class LibraryBrecipe(ConanFile):
    ....

    def build(self):
        ...
        tools.replace_prefix_in_pc_file("mypcfile.pc", "${package_root_path_lib_b}")

    def package(self):
        self.copy(pattern="*.pc", dst="", keep_path=False)
```

Library A recipe (importing and consuming `pc` file):

```
class LibraryArecipe(ConanFile):
    ....

    requires = "libB/1.0@conan/stable, libC/1.0@conan/stable"

    def build(self):

        args = '--define-variable package_root_path_lib_b=%s' % self.deps_cpp_info[
↪"libB"].rootpath
        args += ' --define-variable package_root_path_lib_c=%s' % self.deps_cpp_info[
↪"libC"].rootpath
        pkgconfig_exec = 'pkg-config ' + args

        vars = {'PKG_CONFIG': pkgconfig_exec, # Used by autotools
                'PKG_CONFIG_PATH': "%s:%s" % (self.deps_cpp_info["libB"].rootpath,
                                              self.deps_cpp_info["libC"].rootpath)}

        with tools.environment_append(vars):
            # Call autotools (./configure ./make, will read PKG_CONFIG)
            # Or directly declare the variables:
            self.run('g++ main.cpp $(pkg-config %s libB --libs --cflags) -o main' %␣
↪args)
```

11.9.4 Approach 3: Use *–define-prefix*

If you have available `pkg-config` >= 0.29 and you have only one dependency, you can use directly the `--define-prefix` option to declare a custom `prefix` variable. With this approach you won't need to patch anything, just declare the correct variable.

11.9.5 Approach 4: Use *PKG_CONFIG_$PACKAGE_$VARIABLE*

If you have available `pkg-config` >= 0.29.1 you can manage multiple dependencies declaring N variables with the prefixes:

```
class LibraryArecipe(ConanFile):
    ....

    requires = "libB/1.0@conan/stable, libC/1.0@conan/stable"

    def build(self):

        vars = {'PKG_CONFIG_libB_PREFIX': self.deps_cpp_info["libB"].rootpath,
                'PKG_CONFIG_libC_PREFIX': self.deps_cpp_info["libC"].rootpath,
                'PKG_CONFIG_PATH': "%s:%s" % (self.deps_cpp_info["libB"].rootpath,
                                              self.deps_cpp_info["libC"].rootpath)}

        with tools.environment_append(vars):
            # Call the build system
```

11.9.6 Approach 5: Use the `pkg_config` generator

If you use `package_info()` in libB and libC, and specify all the library names and any other needed flag, you can use the `pkg_config` generator during the `libA`. Those files doesn't need to be patched, because are dynamically generated with the correct path.

So it can be a good solution in case you are building `libA` with a build system that manages `pc files` like *Meson Build* or *AutoTools*:

Meson Build

```
from conans import ConanFile, tools, Meson
import os

class ConanFileToolsTest(ConanFile):
    generators = "pkg_config"
    requires = "LIB_A/0.1@conan/stable"
    settings = "os", "compiler", "build_type"

    def build(self):
        meson = Meson(self)
        meson.configure()
        meson.build()
```

Autotools

```
from conans import ConanFile, tools, Meson
import os

class ConanFileToolsTest(ConanFile):
    generators = "pkg_config"
    requires = "LIB_A/0.1@conan/stable"
    settings = "os", "compiler", "build_type"

    def build(self):
        autotools = AutoToolsBuildEnvironment(self)
```

(continues on next page)

(continued from previous page)

```
        # When using the pkg_config generator, self.build_folder will be added to␣
↪PKG_CONFIG_PATH
        # so pkg_config will be able to locate the generated pc files from the␣
↪requires (LIB_A)
        autotools.configure()
        autotools.make()
```

See also:

Check the *tools.PkgConfig() class*, a wrapper of the `pkg-config` tool that allows to extract flags, library paths, etc for any `pc` file.

11.10 Boost Build

With this generator `boost-build` you can generate a `project-root.jam` file to be used with your Boost Build system.

Check the *generator boost-build*

11.11 QMake

A qmake generator will generate a `conanbuildinfo.pri` file that can be used for your qmake builds.

```
$ conan install . -g qmake
```

Add `conan_basic_setup` to `CONFIG` and include the file in your existing project `.pro` file:

yourproject.pro

```
# ...

CONFIG += conan_basic_setup
include(conanbuildinfo.pri)
```

This will include all the statements in `conanbuildinfo.pri` in your project. Include paths, libraries, defines, etc. will be set up for all requirements you have defined in `conanfile.txt`.

If you'd rather like to manually add the variables for each dependency, you can do so by skipping the CONFIG statement and only include `conanbuildinfo.pri`:

yourproject.pro

```
# ...

include(conanbuildinfo.pri)

# you may now modify your variables manually for each library, such as
# INCLUDEPATH += CONAN_INCLUDEPATH_POCO
```

The `qmake` generator allows multi-configuration packages, i.e. packages that contains both debug and release artifacts. Lets see an example:

11.11.1 Example

There is a complete example in https://github.com/memsharded/qmake_example This project will depend on a multi-configuration (debug/release) "Hello World" package, that should be installed first:

```
$ git clone https://github.com/memsharded/hello_multi_config
$ cd hello_multi_config
$ conan create user/channel
```

This hello package is created with cmake, but that doesn't matter, it can be consumed from a qmake project:

Then, you can get the qmake project and build it, both for debug and release (this example has been tested on linux):

```
$ git clone https://github.com/memsharded/qmake_example
$ cd qmake_example
$ conan install .
$ qmake
$ make
$ ./helloworld
> Hello World Release!
# now lets build the debug one
$ make clean
$ qmake CONFIG+=debug
$ make
$ ./helloworld
> Hello World Debug!
```

See also:

Check the *Reference/Generators/qmake* for the complete reference.

11.12  Premake

From conan 0.9, generator packages are available. Premake4 has experimental support in one of those packages. You can use it as:

```
[requires]
PremakeGen@0.1@memsharded/testing

[generators]
Premake
```

Check the *generator package examples*

https://github.com/memsharded/conan-premake

Link to conan package:

11.13 qbs

Conan provides a **qbs** generator, it will generate a `conanbuildinfo.qbs` file that can be used for your qbs builds.

Add `conanbuildinfo.qbs` as a reference on the project level and a Depends item with the name `conanbuildinfo`:

yourproject.qbs

```
import qbs

Project {
   references: ["conanbuildinfo.qbs"]
   Product {
        type: "application"
        consoleApplication: true
        files: [
            "conanfile.txt",
            "main.cpp",
        ]
        Depends { name: "cpp" }
        Depends { name: "ConanBasicSetup" }
   }
}
```

This will include the product called `ConanBasicSetup` which holds all the necessary settings for all your dependencies.

If you'd rather like to manually add each dependency, just replace `ConanBasicSetup` with the dependency you would like to include. You may also specify multiple dependencies:

yourproject.qbs

```
import qbs

Project {
   references: ["conanbuildinfo.qbs"]
   Product {
        type: "application"
        consoleApplication: true
        files: [
            "conanfile.txt",
            "main.cpp",
        ]
        Depends { name: "cpp" }
        Depends { name: "catch" }
        Depends { name: "Poco" }
   }
}
```

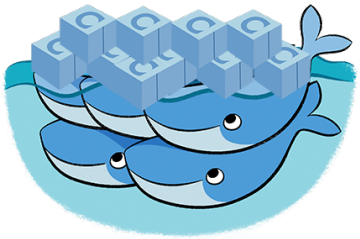
See also:

Check the *Reference/Generators/qbs* section for get more details.

11.14 Meson Build

If you are using **Meson Build** as your library build system, you can use the **Meson** build helper. This helper have `.configure()` and `.build()` methods available to ease the call to meson build system. It also will take automatically the `pc files` of your dependencies when using the *pkg_config generator*.

Check *Building with Meson Build* for more info.



11.15 Docker

You can easily run Conan in a Docker container to build and cross build conan packages.

Check the *'How to use docker to create and cross build C and C++ conan packages'* section to know more.

11.16 Git

Conan uses plain text files, `conanfile.txt` or `conanfile.py`, so it's perfectly suitable for the use of any version control system. We use and highly recommend git.

Check *workflows section* to know more about project layouts that naturally fit version control systems.

11.16.1 Temporary files

Conan generates some files than should not be committed, as `conanbuildinfo.*` and `conaninfo.txt`. These files can change in different computers and are re-generated with the **conan install** command.

However, these files are typically generated in the **build tree** not in the source tree, so they will be naturally disregarded. Just take care in case you have created the **build** folder inside your project (we do this in several examples in this docs). In this case, you should add it to your `.gitignore` file:

.gitignore

```
...
build/
```

11.16.2 Package creators

If you are creating a **conan** package:

- You can use the *url field* to indicate the origin of your package recipe. If you are using an external package recipe, this url should point to the package recipe repository **not** to the external source origin. If a **github** repository is detected, the conan website will link your github issues page from your conan's package page.
- You can use **git** to *obtain your sources* (requires the git client in the path) when creating external package recipes.

11.17 Jenkins

You can use *Jenkins CI* both for:

- Building and testing your project, which manages dependencies with Conan, and probably a conanfile.txt file
- Building and testing conan binary packages for a given conan package recipe (with a conanfile.py) and uploading to a conan remote (Artifactory or conan_server)

There is no need for any special setup for it, just install conan and your build tools in the Jenkins machine and call the needed `conan` commands.

11.17.1 Artifactory and Jenkins integration

If you are using Artifactory you can take advantage of the Jenkins Artifactory Plugin. Check here how to install the plugin and here you can check the full documentation about the DSL.

The Artifactory Jenkins plugin provides a powerful DSL language to call conan, connect with your Artifactory instance, upload and download your packages from Artifactory and manage your build information.

Example: Test your project getting requirements from Artifactory

This is a template to use Jenkins with Artifactory plugin and Conan to retrieve your package from Artifactory server and publish the build information about the downloaded packages to Artifactory.

In this script we assume that we already have all our dependencies in the Artifactory server, and we are building our project that uses **Boost** and **Poco** libraries.

Create a new Jenkins Pipeline task using this script:

```
//Adjust your artifactory instance name/repository and your source code repository
def artifactory_name = "artifactory"
def artifactory_repo = "conan-local"
def repo_url = 'https://github.com/memsharded/example-boost-poco.git'
def repo_branch = 'master'

node {
    def server = Artifactory.server artifactory_name
    def client = Artifactory.newConanClient()
```

(continues on next page)

(continued from previous page)

```
    stage("Get project"){
        git branch: repo_branch, url: repo_url
    }

    stage("Get dependencies and publish build info"){
        sh "mkdir -p build"
        dir ('build') {
          def b = client.run(command: "install ..")
          server.publishBuildInfo b
        }
    }

    stage("Build/Test project"){
        dir ('build') {
          sh "cmake ../ && cmake --build ."
        }
    }
}
```

Stage View

Example: Build a conan package and upload it to Artifactory

In this example we will call conan *test package* command to create a binary packages and then upload it to Artifactory. We also upload the build information:

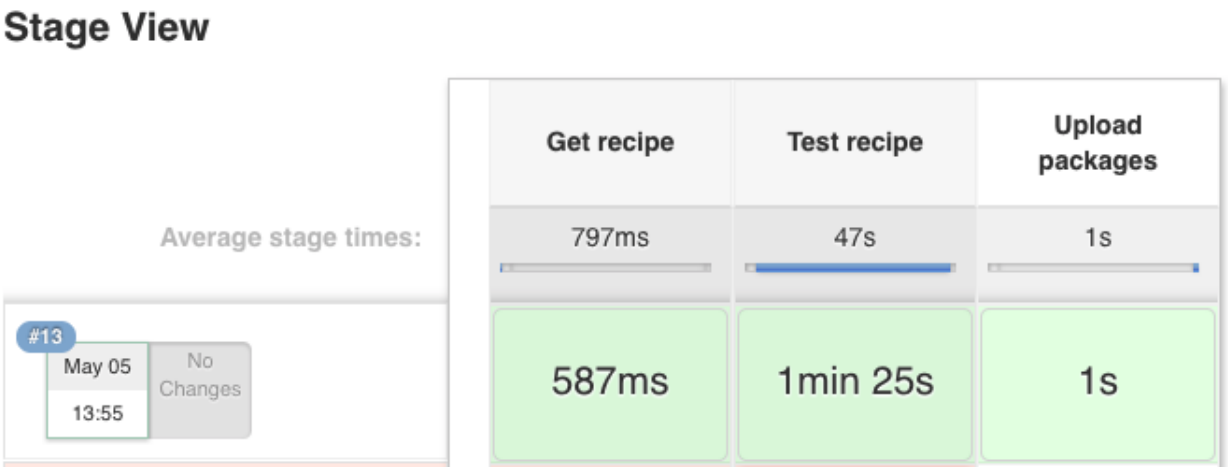
```
def artifactory_name = "artifactory"
def artifactory_repo = "conan-local"
def repo_url = 'https://github.com/lasote/conan-zlib.git'
def repo_branch = "release/1.2.11"

node {
    def server = Artifactory.server artifactory_name
    def client = Artifactory.newConanClient()
    def serverName = client.remote.add server: server, repo: artifactory_repo

    stage("Get recipe"){
        git branch: repo_branch, url: repo_url
    }

    stage("Test recipe"){
        client.run(command: "create")
    }

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



11.18 Travis Ci

You can use Travis CI cloud service to automatically build and test your project in Linux/OSX environments in the cloud. It is free for OSS projects, and offers an easy integration with Github, so builds can be automatically fired in Travis-CI after a **`git push`** to Github.

You can use Travis-CI both for:

- Building and testing your project, which manages dependencies with Conan, and probably a conanfile.txt file
- Building and testing conan binary packages for a given conan package recipe (with a conanfile.py)

11.18.1 Building and testing your project

We are going to use an example with GTest package now, with **Travis CI** support to run the tests.

Clone the project from github:

```
$ git clone https://github.com/lasote/conan-gtest-example
```

Create a `.travis.yml` file and paste this code in it:

```
language: cpp
compiler:
- gcc
install:
# Upgrade GCC
- sudo add-apt-repository ppa:ubuntu-toolchain-r/test -y
- sudo apt-get update -qq
- sudo apt-get install -qq g++-4.9
- sudo update-alternatives --install /usr/bin/gcc gcc /usr/bin/gcc-4.9 60 --slave /
 ↪usr/bin/g++ g++ /usr/bin/g++-4.9

# Install conan
- pip install conan
# Automatic detection of your arch, compiler, etc.
- conan user

script:
# Download dependencies, build, test and create package
- conan create user/channel
```

Travis will install the **conan** tool and will execute the **conan install** command. Then, the **script** section creates the build folder, compiles the project with **cmake** and runs the **tests**.

11.18.2 Creating, testing and uploading conan binary packages

You can use Travis to automate the building of binary packages, which will be created in the cloud after pushing to Github. You can probably setup your own way, but conan has some utilities to help in the process.

The command **conan new** has arguments to create a default working *.travis.yml* file. Other setups might be possible, but for this example we are assuming that you are using github and also uploading your final packages to Bintray. You could follow these steps:

1. First, create an empty github repository, lets call it "hello", for creating a "hello world" package. Github allows to create it with a Readme and .gitignore.
2. Get the credentials User and API Key (remember, Bintray uses the API key as "password", not your main Bintray account password)
3. Create a conan repository in Bintray under your user or organization, and get its URL ("Set me up"). We will call it `UPLOAD_URL`
4. Activate the repo in your Travis account, so it is built when we push changes to it.
5. Under *Travis More Options -> Settings->Environment Variables*, add the `CONAN_PASSWORD` environment variable with the Bintray API Key. If your Bintray user is different from the package user, you can define your Bintray username too, defining the environment variable `CONAN_LOGIN_USERNAME`
6. Clone the repo: `$ git clone <your_repo/hello> && cd hello`
7. Create the package: **conan new Hello/0.1@<user>/testing -t -s -cilg -cis -ciu=UPLOAD_URL** where **user** is your Bintray username.
8. You can inspect the created files: both *.travis.yml*, *.travis/run.sh*, and `.travis/install.sh` and the *build.py* script, that is used by **conan-package-tools** utility to split different builds with different configurations in different travis jobs.
9. You can test locally, before pushing, with **conan test**.
10. Add the changes, commit and push: **git add . && git commit -m "first commit" && git push**.
11. Go to Travis and see the build, with the different jobs.
12. When it finish, go to your Bintray repository, you should see there the uploaded packages for different configurations.
13. Check locally, searching in Bintray: **conan search Hello/0.1@<user>/testing -r=mybintray**.

If something fails, please report an issue in the `conan-package-tools` github repository: https://github.com/conan-io/conan-package-tools

11.19 Appveyor

You can use AppVeyor cloud service to automatically build and test your project in a Windows environment in the cloud. It is free for OSS projects, and offers an easy integration with Github, so builds can be automatically fired in Appveyor after a **git push** to Github.

You can use Appveyor both for:

- Building and testing your project, which manages dependencies with Conan, and probably a conanfile.txt file
- Building and testing conan binary packages for a given conan package recipe (with a conanfile.py)

11.19.1 Building and testing your project

We are going to use an example with GTest package, with **AppVeyor** support to run the tests.

Clone the project from github:

```
$ git clone https://github.com/lasote/conan-gtest-example
```

Create an `appveyor.yml` file and paste this code in it:

```
version: 1.0.{build}
    platform:
      - x64

    install:
      - cmmd: echo "Downloading conan..."
      - cmmd: set PATH=%PATH%;%PYTHON%/Scripts/
      - cmmd: pip.exe install conan
      - cmmd: conan user # Create the conan data directory
      - cmmd: conan --version

    build_script:
      - cmmd: mkdir build
      - cmmd: conan install -o gtest:shared=True
      - cmmd: cd build
      - cmmd: cmake ../ -DBUILD_TEST=TRUE  -G "Visual Studio 14 2015 Win64"
      - cmmd: cmake --build . --config Release

    test_script:
      - cmmd: cd bin
      - cmmd: encryption_test.exe
```

Appveyor will install the **conan** tool and will execute the **conan install** command. Then, the **build_script** section creates the build folder, compiles the project with **cmake** and the section **test_script** runs the **tests**.

11.19.2 Creating, testing and uploading conan binary packages

You can use Appveyor to automate the building of binary packages, which will be created in the cloud after pushing to Github. You can probably setup your own way, but conan has some utilities to help in the process.

The command **`conan new`** has arguments to create a default working *appveyor.yml* file. Other setups might be possible, but for this example we are assuming that you are using GitHub and also uploading your final packages to Bintray. You could follow these steps:

1. First, create an empty github repository, lets call it "hello", for creating a "hello world" package. Github allows to create it with a Readme and .gitignore.
2. Get the credentials User and API Key (remember, Bintray uses the API key as "password", not your main Bintray account password)
3. Create a conan repository in Bintray under your user or organization, and get its URL ("Set me up"). We will call it `UPLOAD_URL`
4. Activate the repo in your Appveyor account, so it is built when we push changes to it.
5. Under *Appveyor Settings->Environment*, add the `CONAN_PASSWORD` environment variable with the Bintray API Key, and encrypt it. If your Bintray user is different from the package user, you can define your Bintray username too, defining the environment variable `CONAN_LOGIN_USERNAME`

6. Clone the repo: `$ git clone <your_repo/hello> && cd hello`
7. Create the package: **conan new Hello/0.1@<user>/testing -t -s -ciw -cis -ciu=UPLOAD_URL** where **user** is your Bintray username
8. You can inspect the created files: both *appveyor.yml* and the *build.py* script, that is used by **conan-package-tools** utility to split different builds with different configurations in different appveyor jobs.
9. You can test locally, before pushing, with **conan create**
10. Add the changes, commit and push: **git add . && git commit -m "first commit" && git push**
11. Go to Appveyor and see the build, with the different jobs.
12. When it finish, go to your Bintray repository, you should see there the uploaded packages for different configurations
13. Check locally, searching in Bintray: **conan search Hello/0.1@<user>/testing -r=mybintray**

If something fails, please report an issue in the `conan-package-tools` github repository: https://github.com/conan-io/conan-package-tools

11.20 Gitlab

You can use Gitlab CI cloud or local service to automatically build and test your project in Linux/OSX/Windows environments. It is free for OSS projects, and offers an easy integration with Gitlab, so builds can be automatically fired in Gitlab CI after a **git push** to Gitlab.

You can use Gitlab CI both for:

- Building and testing your project, which manages dependencies with Conan, and probably a conanfile.txt file
- Building and testing conan binary packages for a given conan package recipe (with a conanfile.py)

11.20.1 Building and testing your project

We are going to use an example with GTest package, with **Gitlab CI** support to run the tests.

Clone the project from github:

```
$ git clone https://github.com/lasote/conan-gtest-example
```

Create a `.gitlab-ci.yml` file and paste this code in it:

```
image: lasote/conangcc63

build:
  before_script:
    # Upgrade Conan version
    - sudo pip install --upgrade conan
    # Automatic detection of your arch, compiler, etc.
```

(continues on next page)

(continued from previous page)

```
    - conan user

  script:
    # Download dependencies, build, test and create package
    - conan create user/channel
```

Gitlab CI will install the **conan** tool and will execute the **conan install** command. Then, the **script** section creates the build folder, compiles the project with **cmake** and runs the **tests**.

11.20.2 Creating, testing and uploading conan binary packages

You can use Gitlab CI to automate the building of binary packages, which will be created in the cloud after pushing to Gitlab. You can probably setup your own way, but conan has some utilities to help in the process.

The command **conan new** has arguments to create a default working `.gitlab-ci.yml` file. Other setups might be possible, but for this example we are assuming that you are using github and also uploading your final packages to Bintray. You could follow these steps:

1. First, create an empty gitlab repository, lets call it "hello", for creating a "hello world" package. Gitlab allows to create it with a Readme, license and .gitignore.
2. Get the credentials User and API Key (remember, Bintray uses the API key as "password", not your main Bintray account password)
3. Create a conan repository in Bintray under your user or organization, and get its URL ("Set me up"). We will call it `UPLOAD_URL`
4. Under your project page, *Settings -> Pipelines -> Add a variable*, add the `CONAN_PASSWORD` environment variable with the Bintray API Key. If your Bintray user is different from the package user, you can define your Bintray username too, defining the environment variable `CONAN_LOGIN_USERNAME`
5. Clone the repo: **git clone <your_repo/hello> && cd hello**.
6. Create the package: **conan new Hello/0.1@<user>/testing -t -s -ciglg -ciglc -cis -ciu=UPLOAD_URL** where **user** is your Bintray username.
7. You can inspect the created files: both *.gitlab-ci.yml* and the *build.py* script, that is used by **conan-package-tools** utility to split different builds with different configurations in different GitLab CI jobs.
8. You can test locally, before pushing, with **conan create** or by GitLab Runner.
9. Add the changes, commit and push: **git add . && git commit -m "first commit" && git push**.
10. Go to Pipelines page and see the pipeline, with the different jobs.
11. When it finish, go to your Bintray repository, you should see there the uploaded packages for different configurations.
12. Check locally, searching in Bintray: **conan search Hello/0.1@<user>/testing -r=mybintray**.

If something fails, please report an issue in the **conan-package-tools** github repository: https://github.com/conan-io/conan-package-tools

11.21 Circle CI

You can use Circle CI cloud to automatically build and test your project in Linux/OSX environments. It is free for OSS projects, and offers an easy integration with Github, so builds can be automatically fired in CircleCI after a `git push` to Github.

You can use CircleCI both for:

- Building and testing your project, which manages dependencies with Conan, and probably a conanfile.txt file
- Building and testing conan binary packages for a given conan package recipe (with a conanfile.py)

11.21.1 Building and testing your project

We are going to use an example with GTest package, with **CircleCI** support to run the tests.

Clone the project from github:

```
$ git clone https://github.com/lasote/conan-gtest-example
```

Create a `.circleci/config.yml` file and paste this code in it:

```
version: 2
gcc-6:
  docker:
    - image: lasote/conangcc6
  steps:
    - checkout
    - run:
        name: Build Conan package
        command: |
          sudo pip install --upgrade conan
          conan user
          conan create . user/channel
workflows:
  version: 2
  build_and_test:
    jobs:
    - gcc-6
```

CircleCI will install the **conan** tool and will execute the **conan create** command. Then, the **script** section creates the build folder, compiles the project with **cmake** and runs the **tests**.

11.21.2 Creating, testing and uploading conan package binaries

You can use CircleCI to automate the building of binary packages, which will be created in the cloud after pushing to Github. You can probably setup your own way, but conan has some utilities to help in the process.

The command `conan new` has arguments to create a default working `.circleci/config.yml` file. Other setups might be possible, but for this example we are assuming that you are using github and also uploading your final packages to Bintray. You could follow these steps:

1. First, create an empty Github repository, lets call it "hello", for creating a "hello world" package. Github allows to create it with a Readme, license and .gitignore.
2. Get the credentials User and API Key (remember, Bintray uses the API key as "password", not your main Bintray account password)
3. Create a conan repository in Bintray under your user or organization, and get its URL ("Set me up"). We will call it `UPLOAD_URL`
4. Under your project page, *Settings -> Pipelines -> Add a variable*, add the `CONAN_PASSWORD` environment variable with the Bintray API Key. If your Bintray user is different from the package user, you can define your Bintray username too, defining the environment variable `CONAN_LOGIN_USERNAME`
5. Clone the repo: `$ git clone <your_repo/hello> && cd hello`
6. Create the package: `$ conan new Hello/0.1@<user>/testing -t -s -ciccg -ciccc -cicco -cis -ciu=UPLOAD_URL` where `user` is your Bintray username
7. You can inspect the created files: both `.circleci/config.yml` and the `build.py` script, that is used by `conan-package-tools` utility to split different builds with different configurations in different GitLab CI jobs.
8. You can test locally, before pushing, with `$ conan create`
9. Add the changes, commit and push: `$ git add . && git commit -m "first commit" && git push`
10. Go to Pipelines page and see the pipeline, with the different jobs.
11. When it finish, go to your Bintray repository, you should see there the uploaded packages for different configurations
12. Check locally, searching in Bintray: `$ conan search Hello/0.1@<user>/testing -r=mybintray`

If something fails, please report an issue in the `conan-package-tools` github repository: https://github.com/conan-io/conan-package-tools

11.22 YouCompleteMe (vim)

If you are a vim user, you are possibly already also a user of YouCompleteMe.

With this generator, you can create the necessary files for your project dependencies, so YouCompleteMe will show symbols from your conan installed dependencies for your project. You only have to add the `ycm` generator to your `conanfile`:

Listing 1: *conanfile.txt*

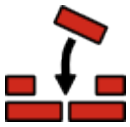
```
[generators]
ycm
```

It will generate a *conan_ycm_extra_conf.py* and a *conan_ycm_flags.json* file in your folder. Those files will be overwritten each time you run **`conan install`**.

In order to make YouCompleteMe work, copy/move *conan_ycm_extra_conf.py* to your project base folder (usually the one containing your `conanfile`) and rename it to *.ycm_extra_conf.py*.

You can (and probably should) edit this file to add your project specific configuration. If your base folder is different from your build folder, link the *conan_ycm_flags.json* from your build folder to your base folder.

```
# from your base folder
$ cp build/conan_ycm_extra_conf.py .ycm_extra_conf.py
$ ln -s build/conan_ycm_flags.json conan_ycm_flags.json
```

11.23  SCons

SCons can be used both to generate and consume conan packages, via the `scons` generator *generator*. The package recipe `build()` method could be similar to:

```
class PkgConan(ConanFile):
    settings = 'os', 'compiler', 'build_type', 'arch'
    requires = 'Hello/1.0@user/stable'
    generators = "scons"

    ...

    def build(self):
        debug_opt = '--debug-build' if self.settings.build_type == 'Debug' else ''
        os.makedirs("build")
        # FIXME: Compiler, version, arch are hardcoded, not parametrized
        with tools.chdir("build"):
            self.run('scons -C {}/src {}'.format(self.source_folder, debug_opt))

    ...
```

The `SConscript` build script can load the generated `SConscript_conan` file that contains the information of the dependencies, and use it to build

```
conan = SConscript('{}/SConscript_conan'.format(build_path_relative_to_sconstruct))
if not conan:
    print 'File `SConscript_conan` is missing.'
    print 'It should be generated by running `conan install`.'
    sys.exit(1)

flags = conan["conan"]
version = flags.pop("VERSION")
env.MergeFlags(flags)
env.Library("hello", "hello.cpp")
```

A complete example, with a `test_package` that also uses SCons is in a github repository, you can try it:

```
$ git clone https://github.com/memsharded/conan-scons-template
$ cd conan-scons-template
$ conan create . demo/testing
> Hello World Release!
$ conan create . demo/testing -s build_type=Debug
> Hello World Debug!
```

11.24 Custom integrations

If you intend to use other build system that has not a built-in generator, you might still be able to do so. There are several options.

- First, search in bintray. Generators can now be created and contributed by users as regular packages, so you can depend on them, use versioning, evolve faster without depending on the conan releases, etc. Check *generator packages*.
- You can use the **text or json generator**. It will generate a text file, simple to read and to parse that you can easily parse with your tools to extract the information.
- Use the **conanfile data model** and access its properties and values, so you can directly call your build system with that information, without requiring to generate a file.
- Write and **create your own generator**. So you can upload it, version and reuse it, as well as share it with your team or community. Check *generator packages* too.

Note: Need help to integrate your build system? Tell us what you need. info@conan.io

11.24.1 Use the JSON generator

Specify the **json** generator in your conanfile:

```
[requires]
fmt/4.1.0@<user>/<stable>
Poco/1.9.0@pocoproject/stable

[generators]
json
```

A file named *conanbuildinfo.json* will be generated. It will contain the information about every dependency:

```
{
  "dependencies":
  [
    {
      "name": "fmt",
      "version": "4.1.0",
      "include_paths": [
        "/path/to/.conan/data/fmt/4.1.0/<user>/<channel>/package/<id>/include"
      ],
      "lib_paths": [
        "/path/to/.conan/data/fmt/4.1.0/<user>/<channel>/package/<id>/lib"
      ],
      "libs": [
        "fmt"
      ],
      "...": "...",
    },
    {
      "name": "Poco",
      "version": "1.7.8p3",
      "...": "..."
    }
```

(continues on next page)

(continued from previous page)

```
  ]
}
```

11.24.2 Use the text generator

Just specify the **txt** generator in your conanfile:

```
[requires]
Poco/1.9.0@pocoproject/stable

[generators]
txt
```

And a file is generated, with the same information as in the case of CMake and gcc, only in a generic, text format, containing the information from the `deps_cpp_info` and `deps_user_info`. Check the conanfile *package_info* method to know more about these objects:

```
[includedirs]
/home/laso/.conan/data/Poco/1.6.1/lasote/stable/package/
↪afafc631e705f7296bec38318b28e4361ab6787c/include
/home/laso/.conan/data/OpenSSL/1.0.2d/lasote/stable/package/
↪154942d8bccb87fbba9157e1daee62e1200e80fc/include
/home/laso/.conan/data/zlib/1.2.8/lasote/stable/package/
↪3b92a20cb586af0d984797002d12b7120d38e95e/include

[libs]
PocoUtil
PocoXML
PocoJSON
PocoMongoDB
PocoNet
PocoCrypto
PocoData
PocoDataSQLite
PocoZip
PocoFoundation
pthread
dl
rt
ssl
crypto
z

[libdirs]
/home/laso/.conan/data/Poco/1.6.1/lasote/stable/package/
↪afafc631e705f7296bec38318b28e4361ab6787c/lib
/home/laso/.conan/data/OpenSSL/1.0.2d/lasote/stable/package/
↪154942d8bccb87fbba9157e1daee62e1200e80fc/lib
/home/laso/.conan/data/zlib/1.2.8/lasote/stable/package/
↪3b92a20cb586af0d984797002d12b7120d38e95e/lib

[bindirs]
/home/laso/.conan/data/Poco/1.6.1/lasote/stable/package/
↪afafc631e705f7296bec38318b28e4361ab6787c/bin
/home/laso/.conan/data/OpenSSL/1.0.2d/lasote/stable/package/
↪154942d8bccb87fbba9157e1daee62e1200e80fc/bin
```

(continues on next page)

(continued from previous page)

```
/home/laso/.conan/data/zlib/1.2.8/lasote/stable/package/
↪3b92a20cb586af0d984797002d12b7120d38e95e/bin

[defines]
POCO_STATIC=ON
POCO_NO_AUTOMATIC_LIBS

[USER_MyRequiredLib1]
somevariable=Some Value
othervar=Othervalue

[USER_MyRequiredLib2]
myvar=34
```

11.24.3 Use conan data model (conanfile.py)

If you are using any other build system you can use conan too. In the `build()` method you can access your settings and build information from your requirements and pass it to your build system. Note, however, that probably is simpler and much more reusable to create a generator to simplify the task for your build system.

```
from conans import ConanFile

class MyProjectWithConan(ConanFile):
   settings = "os", "compiler", "build_type", "arch"
   requires = "Poco/1.9.0@pocoproject/stable"
   ############ IT'S IMPORTANT TO DECLARE THE TXT GENERATOR TO DEAL WITH A GENERIC␣
↪BUILD SYSTEM
   generators = "txt"
   default_options = "Poco:shared=False", "OpenSSL:shared=False"

   def imports(self):
      self.copy("*.dll", dst="bin", src="bin") # From bin to bin
      self.copy("*.dylib*", dst="bin", src="lib") # From lib to bin

   def build(self):
      ############ Without any helper ###########
      # Settings
      print(self.settings.os)
      print(self.settings.arch)
      print(self.settings.compiler)

      # Options
      #print(self.options.my_option)
      print(self.options["OpenSSL"].shared)
      print(self.options["Poco"].shared)

      # Paths and libraries, all
      print("-------- ALL --------------")
      print(self.deps_cpp_info.include_paths)
      print(self.deps_cpp_info.lib_paths)
      print(self.deps_cpp_info.bin_paths)
      print(self.deps_cpp_info.libs)
      print(self.deps_cpp_info.defines)
      print(self.deps_cpp_info.cflags)
      print(self.deps_cpp_info.cppflags)
```

(continues on next page)

(continued from previous page)

```
        print(self.deps_cpp_info.sharedlinkflags)
        print(self.deps_cpp_info.exelinkflags)

        # Just from OpenSSL
        print("--------- FROM OPENSSL -------------")
        print(self.deps_cpp_info["OpenSSL"].include_paths)
        print(self.deps_cpp_info["OpenSSL"].lib_paths)
        print(self.deps_cpp_info["OpenSSL"].bin_paths)
        print(self.deps_cpp_info["OpenSSL"].libs)
        print(self.deps_cpp_info["OpenSSL"].defines)
        print(self.deps_cpp_info["OpenSSL"].cflags)
        print(self.deps_cpp_info["OpenSSL"].cppflags)
        print(self.deps_cpp_info["OpenSSL"].sharedlinkflags)
        print(self.deps_cpp_info["OpenSSL"].exelinkflags)

        # Just from POCO
        print("--------- FROM POCO -------------")
        print(self.deps_cpp_info["Poco"].include_paths)
        print(self.deps_cpp_info["Poco"].lib_paths)
        print(self.deps_cpp_info["Poco"].bin_paths)
        print(self.deps_cpp_info["Poco"].libs)
        print(self.deps_cpp_info["Poco"].defines)
        print(self.deps_cpp_info["Poco"].cflags)
        print(self.deps_cpp_info["Poco"].cppflags)
        print(self.deps_cpp_info["Poco"].sharedlinkflags)
        print(self.deps_cpp_info["Poco"].exelinkflags)


        # self.run("invoke here your configure, make, or others")
        # self.run("basically you can do what you want with your requirements build␣
↪info)


        # Environment variables (from requirements self.env_info objects)
        # are automatically applied in the python ``os.environ`` but can be accesible␣
↪as well:
        print("--------- Globally -------------")
        print(self.env)

        print("--------- FROM MyLib -------------")
        print(self.deps_env_info["MyLib"].some_env_var)


        # User declared variables (from requirements self.user_info objects)
        # are available in the self.deps_user_info object
        print("--------- FROM MyLib -------------")
        print(self.deps_user_info["MyLib"].some_user_var)
```

11.24.4 Create your own generator

There are two ways in which generators can be contributed:

- Forking and adding the new generator in conan codebase. This will be a built-in generator. It might have a much slower release and update cycle, it needs to pass some tests before being accepted, but it has the advantage than no extra things are needed to use that generator (once released in conan)

- Creating a custom *generator package*. You can write a `conanfile.py` and add the custom logic for a generator inside that file, then upload, refer and depend on it as any other package. These generators have to be discovered (search), but they have many advantages: much faster release cycles, independent from the main conan codebase, can be versioned, so backward compatibility and upgrades are much easier.

CHAPTER

TWELVE

HOWTOS

This section shows common solutions and different approaches to typical problems.

12.1 How to package header-only libraries

12.1.1 Without unit tests

Packaging a header only library, without requiring to build and run unit tests for it within conan, can be done with a very simple recipe. Assuming you have the recipe in the source repo root folder, and the headers in a subfolder called `include`, you could do:

```
from conans import ConanFile

class HelloConan(ConanFile):
    name = "Hello"
    version = "0.1"
    # No settings/options are necessary, this is header only
    exports_sources = "include/*"
    no_copy_source = True

    def package(self):
        self.copy("*.h")
```

If you want to package an external repository, you can use the `source()` method to do a clone or download instead of the `exports_sources` fields.

- There is no need for `settings`, as changing them will not affect the final package artifacts
- There is no need for `build()` method, as header-only are not built
- There is no need for a custom `package_info()` method. The default one already adds "include" subfolder to the include path
- `no_copy_source = True` will disable the copy of the source folder to the build directory as there is no need to do so because source code is not modified at all by the `configure()` or `build()` methods.
- Note that this recipe has no other dependencies, settings or options. If it had any of those, it would be very convenient to add the `package_id()` method, to ensure that only one package with always the same ID is create irrespective of the configurations and dependencies:

```
def package_id(self):
    self.info.header_only()
```

Package is created with:

```
$ conan create . user/channel
```

12.1.2 With unit tests

If you want to run the library unit test while packaging, you would need this recipe:

```
from conans import ConanFile, CMake

class HelloConan(ConanFile):
    name = "Hello"
    version = "0.1"
    settings = "os", "compiler", "arch", "build_type"
    exports_sources = "include/*", "CMakeLists.txt", "example.cpp"
    no_copy_source = True

    def build(self): # this is not building a library, just tests
        cmake = CMake(self)
        cmake.configure()
        cmake.build()
        cmake.test()

    def package(self):
        self.copy("*.h")

    def package_id(self):
        self.info.header_only()
```

Tip: If you are *cross building* your **library** or **app** you'll probably need to skip the **unit tests** because your target binary cannot be executed in current building host. To do it you can use *tools.get_env()* in combination with *CONAN_RUN_TESTS* env variable, defined as **False** in profile for cross building and replace `cmake.test()` with:

```
if tools.get_env("CONAN_RUN_TESTS", True):
    cmake.test()
```

Which will use a `CMakeLists.txt` file in the root folder:

```
project(Package CXX)
cmake_minimum_required(VERSION 2.8.12)

include_directories("include")
add_executable(example example.cpp)

enable_testing()
add_test(NAME example
         WORKING_DIRECTORY ${CMAKE_BINARY_DIR}/bin
         COMMAND example)
```

and some `example.cpp` file, which will be our "unit test" of the library:

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

int main() {
```

(continues on next page)

(continued from previous page)

```
    hello();
}
```

- This will use different compilers and versions, as configured by conan settings (in command line or profiles), but will always generate just 1 output package, always with the same ID.
- The necessary files for the unit tests, must be `exports_sources` too (or retrieved from `source()` method)
- If the package had dependencies, via `requires`, it would be necessary to add the `generators = "cmake"` to the package recipe and adding the `conanbuildinfo.cmake` file to the testing CMakeLists.txt:

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

add_executable(example example.cpp)
target_link_libraries(example ${CONAN_LIBS}) # not necessary if dependencies are also␣
↪header-only
```

Package is created with:

```
$ conan create . user/channel
```

Note: This with/without tests is referring to running full unitary tests over the library, which is different to the **`test`** functionality that checks the integrityg of the package. The above examples are describing the approaches for unit-testing the library within the recipe. In either case, it is recommended to have a *test_package* folder, so the **`conan create`** command checks the package once it is created. Check the *packaging getting started guide*

12.2 How to launch conan install from cmake

It is possible to launch **`conan install`** from cmake, which can be convenient for end users, package consumers, that are not creating packages themselves.

This is work under **testing**, please try it and give feedback or contribute. The CMake code to do this task is here: https://github.com/conan-io/cmake-conan

To be able to use it, you can directly download the code from your CMake script:

Listing 1: *CMakeLists.txt*

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

# Download automatically, you can also just copy the conan.cmake file
if(NOT EXISTS "${CMAKE_BINARY_DIR}/conan.cmake")
   message(STATUS "Downloading conan.cmake from https://github.com/conan-io/cmake-
↪conan")
   file(DOWNLOAD "https://raw.githubusercontent.com/conan-io/cmake-conan/master/
↪conan.cmake"
                 "${CMAKE_BINARY_DIR}/conan.cmake")
endif()

include(${CMAKE_BINARY_DIR}/conan.cmake)
```

(continues on next page)

(continued from previous page)

```
conan_cmake_run(REQUIRES Hello/0.1@memsharded/testing
                BASIC_SETUP
                BUILD missing)

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

If you want to use targets, you could do:

```
include(conan.cmake)
conan_cmake_run(REQUIRES Hello/0.1@memsharded/testing
                BASIC_SETUP CMAKE_TARGETS
                BUILD missing)

add_executable(main main.cpp)
target_link_libraries(main CONAN_PKG::Hello)
```

12.3 How to create and reuse packages based on Visual Studio

Conan has different helpers to manage Visual Studio and MSBuild based projects. This how-to illustrates how to put them together to create and consume packages that are purely based on Visual Studio. This how-to is using VS2015, but other versions can be used too.

12.3.1 Creating packages

Start cloning the existing example repository, containing a simple "Hello World" library, and application:

```
$ git clone https://github.com/memsharded/hello_vs
$ cd hello_vs
```

It contains a `src` folder with the source code and a `build` folder with a Visual Studio 2015 solution, containing 2 projects: the HelloLib static library, and the Greet application. Open it:

```
$ build\HelloLib\HelloLib.sln
```

You should be able to select the `Greet` subproject -> `Set as Startup Project`. Then build and run the app with Ctrl+F5. (Debug -> Start Without Debugging)

```
$ Hello World Debug!
# Switch IDE to Release mode, repeat
$ Hello World Release!
```

Because the `hello.cpp` file contains an `#ifdef _DEBUG` to switch between debug and release message.

In the repository, there is already a `conanfile.py` recipe:

```
from conans import ConanFile, MSBuild

class HelloConan(ConanFile):
    name = "Hello"
    version = "0.1"
    license = "MIT"
```

(continues on next page)

(continued from previous page)

```
    url = "https://github.com/memsharded/hello_vs"
    settings = "os", "compiler", "build_type", "arch"
    exports_sources = "src/*", "build/*"

    def build(self):
        msbuild = MSBuild(self)
        msbuild.build("build/HelloLib/HelloLib.sln")

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

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

This recipe is using the *MSBuild() build helper* to build the `sln` project. If our recipe had `requires`, the `MSBUILD` helper will also take care of inject all the needed information from the requirements, as include directories, library names, definitions, flags etc to allow our project to locate the declared dependencies.

The recipe contains also a `test_package` folder with a simple example consuming application. In this example, the consuming application is using cmake to build, but it could also use Visual Studio too. We have left the cmake one because it is the default generated with **`conan new`**, and also to show that packages created from Visual Studio projects can also be consumed with other build systems like CMake.

Once we want to create a package, it is advised to close VS IDE, clean the temporary build files from VS to avoid problems, then create and test the package (here it is using system defaults, assuming they are Visual Studio 14, Release, x86_64):

```
# close VS
$ git clean -xdf
$ conan create . memsharded/testing
...
> Hello World Release!
```

Instead of closing the IDE and running command:*git clean* we could also configure a smarter filter in `exports_sources` field, so temporary build files are not exported into the recipe.

This process can be repeated to create and test packages for different configurations:

```
$ conan create . memsharded/testing -s arch=x86
$ conan create . memsharded/testing -s compiler="Visual Studio" -s compiler.
↪runtime=MDd -s build_type=Debug
$ conan create . memsharded/testing -s compiler="Visual Studio" -s compiler.
↪runtime=MDd -s build_type=Debug -s arch=x86
```

Note: It is not mandatory to specify the `compiler.runtime` setting. If it is not explicitly defined, Conan will automatically use `runtime=MDd` for `build_type==Debug` and `runtime=MD` for `build_type==Release`.

You can list the different created binary packages:

```
$ conan search Hello/0.1@memsharded/testing
```

12.3.2 Uploading binaries

Your locally created packages can already be uploaded to a conan remote. If you created them with the original username "memsharded", as from the git clone, you might want to do a **conan copy** to put them on your own username. Of course, you can also directly use your user name in **conan create**.

Another alternative is to configure the permissions in the remote, to allow uploading packages with different usernames. By default artifactory will do it but conan server won't: permissions must be given in `[write_permissions]` section of `server.conf`.

12.3.3 Reusing packages

To use existing packages directly from Visual Studio, conan provides the `visual_studio` generator. Let's clone an existing "Chat" project, consisting of a ChatLib static library that makes use of the previous "Hello World" package, and a MyChat application, calling the ChatLib library function.

```
$ git clone https://github.com/memsharded/chat_vs
$ cd chat_vs
```

As above, the repository contains a Visual Studio solution in the `build` folder. But if you try to open it, it will fail to load. This is because it is expecting to find a file with the required information about dependencies, so it is necessary to obtain that file first. Just run:

```
$ conan install .
```

You will see that it created two files, a `conaninfo.txt` file, containing the current configuration of dependencies, and a `conanbuildinfo.props` file, containing the Visual Studio properties (like `<AdditionalIncludeDirectories>`), so it is able to find the installed dependencies.

Now you can open the IDE and build and run the app (by the way, the chat function is just calling the `hello()` function two or three times, depending on the build type):

```
$  build\ChatLib\ChatLib.sln
# Switch to Release
# MyChat -> Set as Startup Project
# Ctrl + F5 (Debug -> Run without debugging)
> Hello World Release!
> Hello World Release!
```

If you wish to link with the debug version of Hello package, just install it and change IDE build type:

```
$ conan install . -s build_type=Debug -s compiler="Visual Studio" -s compiler.
↪runtime=MDd
# Switch to Debug
# Ctrl + F5 (Debug -> Run without debugging)
> Hello World Debug!
> Hello World Debug!
> Hello World Debug!
```

Now you can close the IDE and clean the temporary files:

```
# close VS IDE
$ git clean -xdf
```

Again, there is a `conanfile.py` package recipe in the repository, together with a `test_package`. The recipe is almost identical to the above one, just with two minor differences:

```
requires = "Hello/0.1@memsharded/testing"
...
generators = "visual_studio"
```

This will allow us to create and test the package of the ChatLib library:

```
$ conan create . memsharded/testing
> Hello World Release!
> Hello World Release!
```

You can also repeat the process for different build types and architectures.

12.3.4 Other configurations

The above example works as-is for VS2017, because VS supports upgrading from previous versions. The `MSBuild()` already implements such functionality, so building and testing packages with VS2017 can be done.

```
$ conan create . demo/testing -s compiler="Visual Studio" -s compiler.version=15
```

If you have to build for older versions of Visual Studio, it is also possible. In that case, you would probably have different solution projects inside your build folder. Then the recipe only has to select the correct one, something like:

```
def build(self):
    # assuming HelloLibVS12, HelloLibVS14 subfolders
    sln_file = "build/HelloLibVS%s/HelloLib.sln" % self.settings.compiler.version
    msbuild = MSBuild(self)
    msbuild.build(sln_file)
```

Finally, we used just one `conanbuildinfo.props` file, which the solution loaded at a global level. You could also define multiple `conanbuildinfo.props` files, one per configuration (Release/Debug, x86/x86_64), and load them accordingly.

Note: So far, the `visual_studio` generator is single-configuration (packages containing debug or release artifacts, the generally recommended approach), it does not support multi-config packages (packages containing both debug and release artifacts). Please report and provide feedback (submit an issue in github) to request this feature if necessary.

12.4 Creating and reusing packages based on Makefiles

Conan can create packages and reuse them with Makefiles. The `AutoToolsBuildEnvironment` build helper helps with most of the necessary task.

This how-to has been tested in Windows with MinGW and Linux with gcc. It is using static libraries but could be exteded to shared libraries too. The Makefiles surely can be improved they are just an example.

12.4.1 Creating packages

Start cloning the existing example repository, containing a simple "Hello World" library, and application:

```
$ git clone https://github.com/memsharded/conan-example-makefiles
$ cd conan-example-makefiles
$ cd hellolib
```

It contains a *src* folder with the source code and a *conanfile.py* file for creating a package.

Inside the *src* folder, there is *Makefile* to build the static library. This *Makefile* is using standard variables like `$(CPPFLAGS)` or `$(CXX)` to build it:

```
SRC = hello.cpp
OBJ = $(SRC:.cpp=.o)
OUT = libhello.a
INCLUDES = -I.

.SUFFIXES: .cpp

default: $(OUT)

.cpp.o:
    $(CXX) $(INCLUDES) $(CPPFLAGS) $(CXXFLAGS) -c $< -o $@

$(OUT): $(OBJ)
    ar rcs $(OUT) $(OBJ)
```

The *conanfile.py* file uses the `AutoToolsBuildEnvironment` build helper. This helper defines the necessasry environment variables with information from dependencies, as well as other variables to match the current conan settings (like `-m32` or `-m64` based on the conan `arch` setting)

```
from conans import ConanFile, AutoToolsBuildEnvironment
from conans import tools

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

    def build(self):
        with tools.chdir("src"):
            env_build = AutoToolsBuildEnvironment(self)
            # env_build.configure() # use it to run "./configure" if using autotools
            env_build.make()

    def package(self):
        self.copy("*.h", dst="include", src="src")
        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"]
```

With this *conanfile.py* you can create the package:

```
$ conan create . user/testing -s compiler=gcc -s compiler.version=4.9 -s compiler.
↪libcxx=libstdc++
```

12.4.2 Using packages

Now lets move to the application folder:

```
$ cd ../helloapp
```

There you can see also a *src* folder with a *Makefile* creating an executable:

```
SRC = app.cpp
OBJ = $(SRC:.cpp=.o)
OUT = app
INCLUDES = -I.

.SUFFIXES: .cpp

default: $(OUT)

.cpp.o:
    $(CXX) $(CPPFLAGS) $(CXXFLAGS) -c $< -o $@

$(OUT): $(OBJ)
    $(CXX) -o $(OUT)  $(OBJ)  $(LDFLAGS)  $(LIBS)
```

And also a *conanfile.py* very similar to the previous one, in this case adding a `requires` and a `deploy()` method:

```
from conans import ConanFile, AutoToolsBuildEnvironment
from conans import tools

class AppConan(ConanFile):
    name = "App"
    version = "0.1"
    settings = "os", "compiler", "build_type", "arch"
    exports_sources = "src/*"
    requires = "Hello/0.1@user/testing"

    def build(self):
        with tools.chdir("src"):
            env_build = AutoToolsBuildEnvironment(self)
            env_build.make()

    def package(self):
        self.copy("*app", dst="bin", keep_path=False)
        self.copy("*app.exe", dst="bin", keep_path=False)

    def deploy(self):
        self.copy("*", src="bin", dst="bin")
```

Note that in this case, the `AutoToolsBuildEnvironment` will automatically set values to `CPPFLAGS`, `LDFLAGS`, `LIBS`, etc. existing in the *Makefile* with the correct include directories, library names, etc. to properly build and link with the `hello` library contained in the "Hello" package.

As above, we can create the package with:

```
$ conan create . user/testing -s compiler=gcc -s compiler.version=4.9 -s compiler.
↪libcxx=libstdc++
```

There are different ways to run executables contained in packages, like using `virtualrunenv` generators. In this case, as the package has a `deploy()` method, we can use it:

```
$ conan install Hello/0.1user/testing -s compiler=gcc -s compiler.version=4.9 -s␣
↪compiler.libcxx=libstdc++
```

(continues on next page)

(continued from previous page)

```
$ ./bin/app
$ Hello World Release!
```

12.5 How to manage the GCC >= 5 ABI

In the GCC 5.1 release libstdc++ introduced a new library ABI that includes new implementations of `std::string` and `std::list`. These changes were necessary to conform to the 2011 C++ standard which forbids Copy-On-Write strings and requires lists to keep track of their size.

You can choose which ABI to use in your Conan packages by adjusting the `compiler.libcxx`:

- **libstdc++**: Old ABI.
- **libstdc++11**: New ABI.

When Conan create the default profile the first time it runs, adjust the `compiler.libcxx` setting to `libstdc++` for backwards compatibility. If you are using GCC >= 5, your compiler is likely using the new CXX11 ABI by default (libstdc++11).

If you want Conan to use the new ABI, edit the default profile at `~/.conan/profiles/default` adjusting `compiler.libcxx=libstdc++11` or override this setting in the profile you are using.

If you are using the *CMake build helper* or the *AutotoolsBuildEnvironment build helper* Conan will adjust automatically the `_GLIBCXX_USE_CXX11_ABI` flag to manage the ABI.

12.6 Using Visual Studio 2017 - CMake integration

Visual Studio 2017 comes with a CMake integration that allows to just open a folder that contains a *CMakeLists.txt* and Visual will use it to define the project build.

Conan can also be used in this setup to install dependencies. Let's say that we are going to build an application, that depends on an existing conan package called `Hello/0.1@user/testing`. For the purpose of this example, you can quickly create this package typing in your terminal:

```
$ conan new Hello/0.1 -s
$ conan create . user/testing # Default conan profile is Release
$ conan create . user/testing -s build_type=Debug
```

The project we want to develop will be a simple application, with these 3 files in the same folder:

Listing 2: **example.cpp**

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

int main() {
    hello();
    std::cin.ignore();
}
```

Listing 3: **conanfile.txt**

```
[requires]
Hello/0.1@user/testing

[generators]
cmake
```

Listing 4: **CMakeLists.txt**

```
project(Example CXX)
cmake_minimum_required(VERSION 2.8.12)

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

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

If we open Visual Studio 2017 (with CMake support installed), and in the Menu, select "Open Folder" and select the above folder, we will see something like the following error:

```
1> Command line: C:\PROGRAM FILES (X86)\MICROSOFT VISUAL␣
↪STUDIO\2017\COMMUNITY\COMMON7\IDE\COMMONEXTENSIONS\MICROSOFT\CMAKE\CMake\bin\cmake.
↪exe  -G "Ninja" -DCMAKE_INSTALL_PREFIX:PATH="C:\Users\user\CMakeBuilds\df6639d2-
↪3ef2-bc32-abb3-2cd1bdb3c1ab\install\x64-Debug"  -DCMAKE_CXX_COMPILER="C:/Program␣
↪Files (x86)/Microsoft Visual Studio/2017/Community/VC/Tools/MSVC/14.12.25827/bin/
↪HostX64/x64/cl.exe"  -DCMAKE_C_COMPILER="C:/Program Files (x86)/Microsoft Visual␣
↪Studio/2017/Community/VC/Tools/MSVC/14.12.25827/bin/HostX64/x64/cl.exe"  -DCMAKE_
↪BUILD_TYPE="Debug" -DCMAKE_MAKE_PROGRAM="C:\PROGRAM FILES (X86)\MICROSOFT VISUAL␣
↪STUDIO\2017\COMMUNITY\COMMON7\IDE\COMMONEXTENSIONS\MICROSOFT\CMAKE\Ninja\ninja.exe"
↪"C:\Users\user\conanws\visual-cmake"
1> Working directory: C:\Users\user\CMakeBuilds\df6639d2-3ef2-bc32-abb3-
↪2cd1bdb3c1ab\build\x64-Debug
1> -- The CXX compiler identification is MSVC 19.12.25831.0
1> -- Check for working CXX compiler: C:/Program Files (x86)/Microsoft Visual Studio/
↪2017/Community/VC/Tools/MSVC/14.12.25827/bin/HostX64/x64/cl.exe
1> -- Check for working CXX compiler: C:/Program Files (x86)/Microsoft Visual Studio/
↪2017/Community/VC/Tools/MSVC/14.12.25827/bin/HostX64/x64/cl.exe -- works
1> -- Detecting CXX compiler ABI info
1> -- Detecting CXX compiler ABI info - done
1> -- Detecting CXX compile features
1> -- Detecting CXX compile features - done
1> CMake Error at CMakeLists.txt:4 (include):
1>   include could not find load file:
1>
1>     C:/Users/user/CMakeBuilds/df6639d2-3ef2-bc32-abb3-2cd1bdb3c1ab/build/x64-Debug/
↪conanbuildinfo.cmake
```

As expected, our *CMakeLists.txt* is using a `include(${CMAKE_BINARY_DIR}/conanbuildinfo.cmake)`, and that file doesn't exist yet, because conan has not installed the dependencies of this project yet. Visual Studio 2017 uses different build folders for each configuration. In this case, the default configuration at startup is `x64-Debug`. This means that we need to install the dependencies that match this configuration. Assuming that our default profile is using Visual Studio 2017 for x64 (it should typically be the default one created by conan if VS2017 is present), then all we need to specify is the `-s build_type=Debug` setting:

```
$ conan install . -s build_type=Debug -if=C:\Users\user\CMakeBuilds\df6639d2-3ef2-
↪bc32-abb3-2cd1bdb3c1ab\build\x64-Debug
```

(continues on next page)

(continued from previous page)

Now, you should be able to regenerate the CMake project from the IDE, Menu->CMake, build it, select the "example" executable to run, and run it.

Now, lets say that you want to build the Release application. You switch configuration from the IDE, and then the above error happens again. The dependencies for Release mode need to be installed too:

```
$ conan install . -if=C:\Users\user\CMakeBuilds\df6639d2-3ef2-bc32-abb3-
↪2cd1bdb3c1ab\build\x64-Release
```

The process can be extended to x86 (passing `-s arch=x86` in the command line), or to other configurations. For production usage, conan **profiles** are highly recommended.

12.6.1 Using cmake-conan

The **cmake-conan** project in https://github.com/conan-io/cmake-conan is a CMake script that runs an `execute_process` that automatically launches `conan install` to install dependencies. The settings passed in the command line will be deduced from the current CMake configuration, that will match the Visual Studio one. This script can be used to further automate the installation task:

```
project(Example CXX)
cmake_minimum_required(VERSION 2.8.12)

# Download automatically, you can also just copy the conan.cmake file
if(NOT EXISTS "${CMAKE_BINARY_DIR}/conan.cmake")
message(STATUS "Downloading conan.cmake from https://github.com/conan-io/cmake-conan")
    file(DOWNLOAD "https://raw.githubusercontent.com/conan-io/cmake-conan/v0.9/conan.
↪cmake"
                "${CMAKE_BINARY_DIR}/conan.cmake")
endif()

include(${CMAKE_BINARY_DIR}/conan.cmake)

conan_cmake_run(CONANFILE conanfile.txt
                BASIC_SETUP)

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

This code will manage to download the **cmake-conan** CMake script, and use it automatically, calling a `conan install` automatically.

There could be an issue, though, for the `Release` configuration. Internally, the Visual Studio 2017 defines the `configurationType` As `RelWithDebInfo` for `Release` builds. But conan default settings (in the conan *settings.yml* file), only have `Debug` and `Release` defined. It is possible to modify the *settings.yml* file, and add those extra build types. Then you should create the `Hello` package for those settings. And most existing packages, specially in central repositories, are built only for Debug and Release modes.

An easier approach is to change the CMake configuration in Visual: go to the Menu -> CMake -> Change CMake Configuration. That should open the *CMakeSettings.json* file, and there you can change the `configurationType` to `Release`:

```
{
  "name": "x64-Release",
```

(continues on next page)

(continued from previous page)

```
  "generator": "Ninja",
  "configurationType": "Release",
  "inheritEnvironments": [ "msvc_x64_x64" ],
  "buildRoot": "${env.USERPROFILE}\\CMakeBuilds\\${workspaceHash}\\build\\${name}",
  "installRoot": "${env.USERPROFILE}\\CMakeBuilds\\${workspaceHash}\\install\\${name}
↪",
  "cmakeCommandArgs": "",
  "buildCommandArgs": "-v",
  "ctestCommandArgs": ""
}
```

Note that the above CMake code is only valid for consuming existing packages. If you are also creating a package, you would need to make sure the right CMake code is executed, please check https://github.com/conan-io/cmake-conan/blob/master/README.md

12.6.2 Using tasks with tasks.vs.json

Another alternative is using file tasks feature of Visual Studio 2017. This way you can install dependencies by running conan install as task directly in the IDE.

All you need is to right click on your *conanfile.py*-> Configure Tasks (see the link above) and add the following to your *tasks.vs.json*.

Warning: The file *tasks.vs.json* is added to your local *.vs* folder so it is not supposed to be added to your version control system. There is also feature request to improve this process.

```
    {
      "tasks": [
        {
          "taskName": "conan install debug",
          "appliesTo": "conanfile.py",
          "type": "launch",
          "command": "${env.COMSPEC}",
          "args": [
            "conan install ${file} -s build_type=Debug -if C:/Users/user/CMakeBuilds/
↪4c2d87b9-ec5a-9a30-a47a-32ccb6cca172/build/x64-Debug/"
          ]
        },
        {
          "taskName": "conan install release",
          "appliesTo": "conanfile.py",
          "type": "launch",
          "command": "${env.COMSPEC}",
          "args": [
            "conan install ${file} -s build_type=Release -if C:/Users/user/
↪CMakeBuilds/4c2d87b9-ec5a-9a30-a47a-32ccb6cca172/build/x64-Release/"
          ]
        }
      ],
      "version": "0.2.1"
    }
```

Then just right click on your *conanfile.py* and launch your `conan install` and regenerate your *CMakeLists.txt*.

12.7 How to manage C++ standard

Warning: This feature is experimental

The setting representing the C++ standard is `cppstd`. The detected default profile doesn't set any value for the `cppstd` setting.

The consumer can specify it in a *profile* or with the `-s` parameter:

```
conan install . -s cppstd=gnu14
```

This setting will only be applied to the recipes specifying `cppstd` in the `settings` field:

```
class LibConan(ConanFile):
    name = "lib"
    version = "1.0"
    settings = "cppstd", "os", "compiler", "build_type", "arch"
```

Valid values for `compiler=Visual Studio`:

VALUE	DESCRIPTION
14	C++ 14
17	C++ 17

Valid values for other compilers:

VALUE	DESCRIPTION
98	C++ 98
gnu98	C++ 98 with GNU extensions
11	C++ 11
gnu11	C++ 11 with GNU extensions
14	C++ 14
gnu14	C++ 14 with GNU extensions
17	C++ 17
gnu17	C++ 17 with GNU extensions

12.7.1 Build helpers

When the `cppstd` setting is declared in the recipe and the consumer specify a value for it:

- The *CMake* build helper will set the `CONAN_CMAKE_CXX_STANDARD` and `CONAN_CMAKE_CXX_EXTENSIONS` definitions, that will be converted to the corresponding CMake variables to activate the standard automatically with the `conan_basic_setup()` macro.
- The *AutotoolsBuildEnvironment* build helper will adjust the needed flag to `CXXFLAGS` automatically.
- The *MSBuild/VisualStudioBuildEnvironment* build helper will adjust the needed flag to `CL` env var automatically.

12.7.2 Package compatibility

By default Conan will detect the default standard of your compiler to not generate different binary packages. For example, you already built some `gcc > 6.1` packages, where the default std is `gnu14`. If you introduce the `cppstd`

setting in your recipes and specify the `gnu14` value, Conan won't generate new packages, because it was already the default of your compiler.

Note: Check the *package_id()* reference to know more.

12.8 How to use docker to create and cross build C and C++ conan packages

With Docker, you can run different virtual Linux operating systems in a Linux, Mac OSX or Windows machine. It is useful to reproduce build environments, for example to automate CI processes. You can have different images with different compilers or toolchains and run containers every time is needed.

In this section you will find a *list of pre-built images* with common build tools and compilers as well as Conan installed.

12.8.1 Using conan inside a container

```
$ docker run -it --rm lasote/conangcc7 /bin/bash
```

Note: Use `sudo` when needed to run `docker`.

The previous code will run a shell in container. We have specified:

- **`-it`**: Keep STDIN open and allocate a pseudo-tty, in other words, we want to type in the container because we are opening a bash.
- **`--rm`**: Once the container exits, remove the container. Helps to keep clean or hard drive.
- **`lasote/conangcc7`**: Image name, check the *available docker images*.
- **`/bin/bash`**: The command to run

Now we are running on the conangcc7 container we can use Conan normally. In the following example we are creating a package from the recipe by cloning the repository, for OpenSSL. It is always recommended to upgrade conan from pip first:

```
$ sudo pip install conan --upgrade # We make sure we are running the latest Conan␣
↪version
$ git clone https://github.com/conan-community/conan-openssl
$ cd conan-openssl
$ conan create . user/channel
```

12.8.2 Sharing a local folder with a docker container

You can share a local folder with your container, for example a project:

```
$ git clone https://github.com/conan-community/conan-openssl
$ cd conan-openssl
$ docker run -it -v$(pwd):/home/conan/project --rm lasote/conangcc7 /bin/bash
```

- `v$(pwd):/home/conan/project`: We are mapping the current directory (conan-openssl) to the container `/home/conan/project` directory, so anything we change in this shared folder, will be really changed in our host machine.

```
# Now we are running on the conangcc7 container
$ sudo pip install conan --upgrade # We make sure we are running the latest Conan␣
↪version
$ cd project
$ conan create . user/channel --build missing
$ conan remote add myremote http://some.remote.url
$ conan upload "*" -r myremote --all
```

12.8.3 Using the images to cross-build packages

You can use the *images* `-i386`, `-armv7` and `-armv7gh` to cross build conan packages.

The `armv7` images have a cross toolchain for linux ARM installed, and declared as main compiler with the environment variables `CC` and `CXX`. Also, the default Conan profile (`~/.conan/profiles/default`) is adjusted to declare the correct arch (`armv7` / `armv7hf`).

Cross-building and uploading a package along with all its missing dependencies for `Linux/armv7hf` is done in few steps:

```
$ git clone https://github.com/conan-community/conan-openssl
$ cd conan-openssl
$ docker run -it -v$(pwd):/home/conan/project --rm lasote/conangcc49-armv7hf /bin/bash

# Now we are running on the conangcc49-armv7hf container
# The default profile is automatically adjusted to armv7hf
$ cat ~/.conan/profiles/default

[settings]
os=Linux
os_build=Linux
arch=armv7hf
arch_build=x86_64
compiler=gcc
compiler.version=4.9
compiler.libcxx=libstdc++
build_type=Release
[options]
[build_requires]
[env]

$ sudo pip install conan --upgrade # We make sure we are running the latest Conan␣
↪version
$ cd project

$ conan create . user/channel --build missing
$ conan remote add myremoteARMV7 http://some.remote.url
$ conan upload "*" -r myremoteARMV7 --all
```

12.8.4 Available docker images

GCC images

Version	Target Arch
lasote/conangcc49 (GCC 4.9)	x86_64
lasote/conangcc49-i386 (GCC 4.9)	x86
lasote/conangcc49-armv7 (GCC 4.9)	armv7
lasote/conangcc49-armv7hf (GCC 4.9)	armv7hf
lasote/conangcc5-armv7 (GCC 5)	armv7
lasote/conangcc5-armv7hf (GCC 5)	armv7hf
lasote/conangcc5 (GCC 5)	x86_64
lasote/conangcc5-i386 (GCC 5)	x86
lasote/conangcc5-armv7 (GCC 5)	armv7
lasote/conangcc5-armv7hf (GCC 5)	armv7hf
lasote/conangcc6 (GCC 6)	x86_64
lasote/conangcc6-i386 (GCC 6)	x86
lasote/conangcc6-armv7 (GCC 6)	armv7
lasote/conangcc6-armv7hf: (GCC 6)	armv7hf
lasote/conangcc7-i386 (GCC 7)	x86
lasote/conangcc7 (GCC 7)	x86_64
lasote/conangcc7-armv7 (GCC 7)	armv7
lasote/conangcc7-armv7hf (GCC 7)	armv7hf

Clang images

Version	Target Arch
lasote/conanclang38 (Clang 3.8)	x86_64
lasote/conanclang39-i386 (Clang 3.9)	x86
lasote/conanclang39 (Clang 3.9)	x86_64
lasote/conanclang40-i386 (Clang 4)	x86
lasote/conanclang40 (Clang 4)	x86_64
lasote/conanclang50-i386 (Clang 5)	x86
lasote/conanclang50 (Clang 5)	x86_64

The Dockerfiles for all these images can be found here.

12.9 How to reuse Python code in recipes

First, if you feel that you are repeating a lot of Python code, and that repeated code could be useful for other Conan users, please propose it in a github issue.

There are several ways to handle Python code reuse in package recipes:

- To put common code in files, as explained *below*. This code has to be exported into the recipe itself.
- To create a Conan package with the common python code, and then `require` it from the recipe.

This howto explains the latter.

12.9.1 A basic Python package

Let's begin with a simple python package, a "hello world" functionality that we want to package and reuse:

```
def hello():
    print("Hello World from Python!")
```

To create a package, all we need to do is create the following layout:

```
-| hello.py
 | __init__.py
 | conanfile.py
```

The `__init__.py` is blank. It is not necessary to compile code, so the package recipe `conanfile.py` is quite simple:

```
from conans import ConanFile

class HelloPythonConan(ConanFile):
    name = "HelloPy"
    version = "0.1"
    exports = '*'
    build_policy = "missing"

    def package(self):
        self.copy('*.py')

    def package_info(self):
        self.env_info.PYTHONPATH.append(self.package_folder)
```

The `exports` will copy both the `hello.py` and the `__init__.py` into the recipe. The `package()` method is also obvious: to construct the package just copy the python sources.

The `package_info()` adds the current package folder to the `PYTHONPATH` conan environment variable. It will not affect the real environment variable unless the end user wants it.

It can be seen that this recipe would be practically the same for most python packages, so it could be factored in a `PythonConanFile` base class to further simplify it (open a feature request, or better a pull request :))

With this recipe, all we have to do is:

```
$ conan export . memsharded/testing
```

Of course if you want to share the package with your team, you can **`conan upload`** it to a remote server. But to create and test the package, we can do everything locally.

Now the package is ready for consumption. In another folder, we can create a *conanfile.txt* (or a *conanfile.py* if we prefer):

```
[requires]
HelloPy/0.1@memsharded/testing
```

And install it with the following command:

```
$ conan install . -g virtualenv
```

Creating the above `conanfile.txt` might be unnecessary for this simple example, as you can directly run **`conan install HelloPy/0.1@memsharded/testing -g virtualenv`**, however, using the file is the canonical way.

The specified `virtualenv` generator will create an `activate` script (in Windows *activate.bat*), that basically contains the environment, in this case, the `PYTHONPATH`. Once we activate it, we are able to find the package in the path and use it:

```
$ activate
$ python
Python 2.7.12 (v2.7.12:d33e0cf91556, Jun 27 2016, 15:19:22) [MSC v.1500 32 bit␣
↪(Intel)] on win32
...
>>> import hello
>>> hello.hello()
Hello World from Python!
>>>
```

The above shows an interactive session, but you can import also the functionality in a regular python script.

12.9.2 Reusing python code in your recipes

Requiring a python conan package

As the conan recipes are python code itself, it is easy to reuse python packages in them. A basic recipe using the created package would be:

```
from conans import ConanFile

class HelloPythonReuseConan(ConanFile):
    requires = "HelloPy/0.1@memsharded/testing"

    def build(self):
        from hello import hello
        hello()
```

The `requires` section is just referencing the previously created package. The functionality of that package can be used in several methods of the recipe: `source()`, `build()`, `package()` and `package_info()`, i.e. all of the methods used for creating the package itself. Note that in other places it is not possible, as it would require the dependencies of the recipe to be already retrieved, and such dependencies cannot be retrieved until the basic evaluation of the recipe has been executed.

```
$ conan install .
...
$ conan build .
Hello World from Python!
```

Sharing a python module

Another approach is sharing a python module and exporting within the recipe.

Lets write for example a `msgs.py` file and put it besides the `conanfile.py`:

```
def build_msg(output):
    output.info("Building!")
```

And then the main `conanfile.py` would be:

```
from conans import ConanFile
from msgs import build_msg

class ConanFileToolsTest(ConanFile):
```

(continues on next page)

(continued from previous page)

```
    name = "test"
    version = "1.9"
    exports = "msgs.py"  # Important to remember!

    def build(self):
        build_msg(self.output)
        # ...
```

It is important to note that such `msgs.py` file **must be exported** too when exporting the package, because package recipes must be self-contained.

The code reuse can also be done in the form of a base class, something like a file `base_conan.py`

```
from conans import ConanFile

class ConanBase(ConanFile):
    # common code here
```

And then:

```
from conans import ConanFile
from base_conan import ConanBase

class ConanFileToolsTest(ConanBase):
    name = "test"
    version = "1.9"
    exports = "base_conan.py"
```

12.10 How to create and share a custom generator with generator packages

There are several built-in generators, like `cmake`, `visual_studio`, `xcode`... But what if your build system is not included? Or maybe the existing built-in generators doesn't satisfy your needs. There are several options:

- Use the `txt` generator, that generates a plain text file easy to parse, which you might be able to use.
- Use `conanfile.py` data, and for example in the `build()` method, access that information directly and generate a file or call directly your system
- Fork the conan codebase and write a built-in generator. Please make a pull request if possible to contribute it to the community.
- Write a custom generator in a `conanfile.py` and manage it as a package. You can upload it to your own server and share with your team, or share with the world uploading it to bintray. You can manage it as a package, you can version it, overwrite it, delete it, create channels (testing/stable...), and the most important: bring it to your projects as a regular dependency.

This **how to** will show you how to do the latest one. We will build a generator for **premake** (https://premake.github.io/) build system:

12.10.1 Creating a custom generator

Basically a generator is a class that extends `Generator` and implements two properties: `filename`, which will be the name of the file that will be generated, and `content` with the contents of that file. The **name of the generator**

itself will be taken from the class name:

```
class MyGeneratorName(Generator):
    @property
    def filename(self):
        return "mygenerator.file"

    @property
    def content(self):
        return "whatever contents the generator produces"
```

This class is just included in a `conanfile.py` that must contain also a `Conanfile` class that implements the package itself, with the name of the package, the version, etc. This class typically has no `source()`, `build()`, `package()`, and even the `package_info()` method is overriden as it doesn't have to define any include paths or library paths.

If you want to create a generator that creates more than one file, you can leave the `filename()` empty, and return a dictionary of filenames->contents in the `content()` method:

```
class MultiGenerator(Generator):

    @property
    def content(self):
        return {"filename1.txt": "contents of file1",
                "filename2.txt": "contents of file2"}  # any number of files

    @property
    def filename(self):
        pass
```

Once, it is defined in the `conanfile.py` you can treat is as a regular package, typically you will `export` it first to your local cache, test it, and once it is working fine, you would `upload` it to a server.

12.10.2 Premake generator example

Create a project (the best is a git repository):

```
$ mkdir conan-premake && cd conan-premake
```

Then, write in it the following **conanfile.py**:

```
from conans.model import Generator
from conans import ConanFile

class PremakeDeps(object):
    def __init__(self, deps_cpp_info):
        self.include_paths = ",\n".join('"%s"' % p.replace("\\", "/")
                                        for p in deps_cpp_info.include_paths)
        self.lib_paths = ",\n".join('"%s"' % p.replace("\\", "/")
                                    for p in deps_cpp_info.lib_paths)
        self.bin_paths = ",\n".join('"%s"' % p.replace("\\", "/")
                                    for p in deps_cpp_info.bin_paths)
        self.libs = ", ".join('"%s"' % p for p in deps_cpp_info.libs)
        self.defines = ", ".join('"%s"' % p for p in deps_cpp_info.defines)
        self.cppflags = ", ".join('"%s"' % p for p in deps_cpp_info.cppflags)
        self.cflags = ", ".join('"%s"' % p for p in deps_cpp_info.cflags)
```

(continues on next page)

(continued from previous page)

```
        self.sharedlinkflags = ", ".join('"%s"' % p for p in deps_cpp_info.
↪sharedlinkflags)
        self.exelinkflags = ", ".join('"%s"' % p for p in deps_cpp_info.exelinkflags)

        self.rootpath = "%s" % deps_cpp_info.rootpath.replace("\\", "/")

class Premake(Generator):
    @property
    def filename(self):
        return "conanpremake.lua"

    @property
    def content(self):
        deps = PremakeDeps(self.deps_build_info)

        template = ('conan_includedirs{dep} = {{{deps.include_paths}}}\n'
                    'conan_libdirs{dep} = {{{deps.lib_paths}}}\n'
                    'conan_bindirs{dep} = {{{deps.bin_paths}}}\n'
                    'conan_libs{dep} = {{{deps.libs}}}\n'
                    'conan_cppdefines{dep} = {{{deps.defines}}}\n'
                    'conan_cppflags{dep} = {{{deps.cppflags}}}\n'
                    'conan_cflags{dep} = {{{deps.cflags}}}\n'
                    'conan_sharedlinkflags{dep} = {{{deps.sharedlinkflags}}}\n'
                    'conan_exelinkflags{dep} = {{{deps.exelinkflags}}}\n')

        sections = ["#!lua"]
        all_flags = template.format(dep="", deps=deps)
        sections.append(all_flags)
        template_deps = template + 'conan_rootpath{dep} = "{deps.rootpath}"\n'

        for dep_name, dep_cpp_info in self.deps_build_info.dependencies:
            deps = PremakeDeps(dep_cpp_info)
            dep_name = dep_name.replace("-", "_")
            dep_flags = template_deps.format(dep="_" + dep_name, deps=deps)
            sections.append(dep_flags)

        return "\n".join(sections)


class MyCustomGeneratorPackage(ConanFile):
    name = "PremakeGen"
    version = "0.1"
    url = "https://github.com/memsharded/conan-premake"
    license = "MIT"

    def build(self):
        pass

    def package_info(self):
        self.cpp_info.includedirs = []
        self.cpp_info.libdirs = []
        self.cpp_info.bindirs = []
```

This is a full working example. Note the `PremakeDeps` class as a helper. The generator is creating premake information for each individual library separately, then also an aggregated information for all dependencies. This `PremakeDeps` wraps a single item of such information.

Note the **name of the package** will be **PremakeGen/0.1@user/channel** as that is the name given to it, while the generator name is **Premake**. You can give the package any name you want, even matching the generator name if desired.

You `export` the package recipe to the local cache, so it can be used by other projects as usual:

```
$ conan export memsharded/testing
```

12.10.3 Using the generator

Let's create a test project that uses this generator, and also an existing library conan package, we will use the simple "Hello World" package we already created before:

```
$ cd ..
$ mkdir premake-project && cd premake-project
```

Now put the following files inside. Note the `PremakeGen@0.1@memsharded/testing` package reference in conanfile.txt.

conanfile.txt

```
[requires]
Hello/0.1@memsharded/testing
PremakeGen@0.1@memsharded/testing

[generators]
Premake
```

main.cpp

```
#include "hello.h"

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

premake4.lua

```
#!lua

require 'conanpremake'

-- A solution contains projects, and defines the available configurations
solution "MyApplication"
   configurations { "Debug", "Release" }
   includedirs { conan_includedirs }
   libdirs { conan_libdirs }
   links { conan_libs }
   -- A project defines one build target
   project "MyApplication"
      kind "ConsoleApp"
      language "C++"
      files { "**.h", "**.cpp" }

      configuration "Debug"
         defines { "DEBUG" }
```

(continues on next page)

(continued from previous page)

```
        flags { "Symbols" }

    configuration "Release"
        defines { "NDEBUG" }
        flags { "Optimize" }
```

Let's install the requirements and build the project:

```
$ conan install  -s compiler=gcc -s compiler.version=4.9 -s compiler.libcxx=libstdc++␣
↪--build
$ premake4 gmake
$ make (or mingw32-make if in windows-mingw)
$ ./MyApplication
Hello World!
```

Now, everything works, so you might want to share your generator:

```
$ conan upload PremakeGen/0.1@memsharded/testing
```

Note: This is a regular conan package. You could for example embed this example in a *test_package* folder, create a *conanfile.py* that invokes premake4 in the build() method, and use **conan test** to automatically test your custom generator with a real project.

12.11 How to manage shared libraries

The shared libraries, *.DLL* in windows, *.dylib* in OSX and *.so* in Linux, are loaded at runtime, that means that the application executable needs to know where are the required shared libraries when it runs.

On Windows, the dynamic linker, will search in the same directory then in the *PATH* directories. On OSX, it will search in the directories declared in *DYLD_LIBRARY_PATH* as on Linux will use the *LD_LIBRARY_PATH*.

Furthermore in OSX and Linux there is another mechanism to locate the shared libraries: The RPATHs.

12.11.1 Manage Shared Libraries with Environment Variables

The shared libraries, are loaded at runtime. The application executable needs to know where to find the required shared libraries when it runs.

Depending on the operating system, we can use environment variables to help the dynamic linker to find the shared libraries:

OPERATING SYSTEM	ENVIRONMENT VARIABLE
WINDOWS	PATH
LINUX	LD_LIBRARY_PATH
OSX	DYLD_LIBRARY_PATH

If your package recipe (A) is generating shared libraries you can declare the needed environment variables pointing to the package directory. This way, any other package depending on (A) will automatically have the right environment variable set, so they will be able to locate the (A) shared library.

Similarly if you use the *virtualenv generator* and you activate it, you will get the paths needed to locate the shared libraries in your terminal.

Example

We are packaging a tool called `toolA` with a library and an executable that, for example, compress data.

The package offers two flavors, shared library or static library (embedded in the executable of the tool and available to link with). You can use the `toolA` package library to develop another executable or library or you can just use the executable provided by the package. In both cases, if you choose to install the *shared* package of `toolA` you will need to have the shared library available.

```
import os
from conans import tools, ConanFile

class ToolA(ConanFile):
    ....
    name = "toolA"
    version = "1.0"
    options = {"shared": [True, False]}
    default_options = "shared=False"


    def build(self):
        # build your shared library

    def package(self):
        # Copy the executable
        self.copy(pattern="toolA*", dst="bin", keep_path=False)

        # Copy the libraries
        if self.options.shared:
            self.copy(pattern="*.dll", dst="bin", keep_path=False)
            self.copy(pattern="*.dylib", dst="lib", keep_path=False)
            self.copy(pattern="*.so*", dst="lib", keep_path=False)
        else:
            ...


    def package_info(self):
        self.env_info.PATH.append(os.path.join(self.package_folder, "bin"))
        if self.options.shared:
            self.env_info.LD_LIBRARY_PATH.append(os.path.join(self.package_folder,
↪"lib"))
            self.env_info.DYLD_LIBRARY_PATH.append(os.path.join(self.package_folder,
↪"lib"))
```

Using the tool from a different package

If we are creating now a package that uses the `ToolA` executable to compress some data. You can call directly to the `toolA.exe`, the required environment variables to locate both the executable and the shared libraries are automatically available:

```
import os
from conans import tools, ConanFile
```

(continues on next page)

(continued from previous page)

```
class PackageB(ConanFile):
    ....
    name = "packageB"
    version = "1.0"
    requires = "toolA/1.0@myuser/stable"


    def build(self):
        ...
        # we can call directly the ``toolA`` executable. the shared library will be
↪located too
        exe_name = "toolA.exe" if self.settings.os == "Windows" else "toolA"
        self.run("%s --someparams" % exe_name)
        ...
```

Building an application using the shared library from `toolA`

As we are building a final application, probably we will want to distribute it together with the shared library from the `toolA`, so we can use the *Imports* to import the required shared libraries to our user space.

conanfile.txt

```
[requires]
toolA/1.0@myuser/stable

[generators]
cmake

[options]
toolA:shared=True

[imports]
bin, *.dll -> ./bin # Copies all dll files from packages bin folder to my "bin" folder
lib, *.dylib* -> ./bin # Copies all dylib files from packages lib folder to my "bin"
↪folder
lib, *.so* -> ./bin # Copies all dylib files from packages lib folder to my "bin"
↪folder
```

In the terminal window and build the project:

```
$ mkdir build && cd build
$ conan install ..
$ cmake .. -G "Visual Studio 14 Win64"
$ cmake --build . --config Release
$ cd bin && mytool
```

The previous example will work only in Windows and OSX (changing the CMake generator), because the dynamic linker will look in the current directory (the binary directory) where we copied the shared libraries too.

In Linux you still need to set the `LD_LIBRARY_PATH`, or in OSX, the `DYLD_LIBRARY_PATH`:

```
$ cd bin && LD_LIBRARY_PATH=$(pwd) && ./mytool
```

Using shared libraries from dependencies

If you are executing something that depends on shared libraries belonging to your dependencies, such shared libraries have to be found at runtime. In Windows, it is enough if the package added its binary folder to the system `PATH`. In Linux and OSX, it is necessary that the `LD_LIBRARY_PATH` and `DYLD_LIBRARY_PATH` environment variables are used.

Security restrictions might apply in OSX (read this thread), so the `DYLD_LIBRARY_PATH` environment variable is not directly transferred to the child process. In that case, you have to use it explicitely in your conanfile.py:

```
def test(self):
    # self.run('./myexe") # won't work, even if 'DYLD_LIBRARY_PATH' is in the env
    self.run('DYLD_LIBRARY_PATH=%s ./myexe" % os.environ['DYLD_LIBRARY_PATH'])
```

Using the virtualenv generator

We could also use a *virtualenv generator* to get the `toolA` executable available:

conanfile.txt

```
[requires]
toolA/1.0@myuser/stable

[options]
toolA:shared=True

[generators]
virtualenv
```

In the terminal window:

```
conan install .
source activate
toolA --someparams
```

Using the virtualrunenv generator

Even if `toolA` doesn't declare the variables in the `package_info` method, you can use the *virtualrunenv generator*. It will set automatically the environment variables poiting to the "lib" and "bin" folders.

conanfile.txt

```
[requires]
toolA/1.0@myuser/stable

[options]
toolA:shared=True

[generators]
virtualenv
```

In the terminal window:

```
conan install .
source activate
toolA --someparams
```

12.11.2 Manage RPATHs

The **rpath** is encoded inside dynamic libraries and executables and helps the linker to find its required shared libraries.

If we have an executable, **my_exe**, that requires a shared library, **shared_lib_1**, and **shared_lib_1**, in turn, requires another **shared_lib_2**.

So the **rpaths** values are:

File	rpath
my_exe	/path/to/shared_lib_1
shared_lib_1	/path/to/shared_lib_2
shared_lib_2	

In **linux** if the linker doesn't find the library in **rpath**, it will continue the search in **system defaults paths** (LD_LIBRARY_PATH... etc) In OSX, if the linker detects an invalid **rpath** (the file does not exist there), it will fail.

Default Conan approach

The consumer project of dependencies with shared libraries needs to import them to the executable directory to be able to run it:

conanfile.txt

```
[requires]
Poco/1.9.0@pocoproject/stable

[imports]
bin, *.dll -> ./bin # Copies all dll files from packages bin folder to my "bin" folder
lib, *.dylib* -> ./bin # Copies all dylib files from packages lib folder to my "bin"␣
↪folder
```

On **Windows** this approach works well, importing the shared library to the directory containing your executable is a very common procedure.

On **Linux** there is an additional problem, the dynamic linker doesn't look by default in the executable directory, and you will need to adjust the *LD_LIBRARY_PATH* environment variable like this:

```
LD_LIBRARY_PATH=$(pwd) && ./mybin
```

On **OSX** if absolute rpaths are hardcoded in an executable or shared library and they don't exist the executable will fail to run. This is the most common problem when we reuse packages in a different environment from where the artifacts have been generated.

So, for **OSX**, conan, by default when you build your library with **CMake**, the rpaths will be generated without any path:

File	rpath
my_exe	shared_lib_1.dylib
shared_lib_1.dylib	shared_lib_2.dylib
shared_lib_2.dylib	

The `conan_basic_setup()` macro will set the `set(CMAKE_SKIP_RPATH 1)` in OSX.

You can skip this default behavior by passing the `KEEP_RPATHS` parameter to the `conan_basic_setup` macro:

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

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

If you are using `autotools` conan won't auto-adjust the rpaths behavior, if you want to follow this default behavior probably you will need to replace the `install_name` in the **configure** or **MakeFile** generated files in your recipe to not use $rpath:

```
replace_in_file("./configure", r"-install_name \$rpath/", "-install_name ")
```

Different approaches

You can adjust the **rpaths** in the way that adapts better to your needs.

If you are using `CMake` take a look to the CMake RPATH handling guide.

Remember to pass the `KEEP_RPATHS` variable to the `conan_basic_setup`:

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

Then, you could, for example, use the `@executable_path` in OSX and `$ORIGIN` in Linux to adjust a relative path from the executable:

```
if (APPLE)
    set(CMAKE_INSTALL_RPATH "@executable_path/../lib")
else()
    set(CMAKE_INSTALL_RPATH "$ORIGIN/../lib")
endif()
```

You can use this imports statements in the consumer project:

```
[requires]
Poco/1.9.0@pocoproject/stable

[imports]
bin, *.dll -> ./bin # Copies all dll files from packages bin folder to my "bin" folder
lib, *.dylib* -> ./lib # Copies all dylib files from packages lib folder to my "lib"␣
↪folder
lib, *.so* -> ./lib # Copies all so files from packages lib folder to my "lib" folder
```

And your finally application can follow this layout:

```
bin
 |_____ my_executable
 |_____ mylib.dll
 |
lib
 |_____ libmylib.so
 |_____ libmylib.dylib
```

You could move the entire application folder to any location and the shared libraries will be located correctly.

12.12 How to reuse cmake install for package() method

It is possible that your project's *CMakeLists.txt* has already defined some functionality that extracts the artifacts (headers, libraries, binaries) from the build and source folder to a predetermined place.

The conan `package()` method does exactly that: it defines which files have to be copied from the build folder to the package folder.

If you want to reuse that functionality, you can do it with cmake.

Invoke cmake with `CMAKE_INSTALL_PREFIX` using the `package_folder` variable. If the `cmake install` target correctly copies all the required libraries, headers, etc. to the `package_folder`, then the `package()` method is not required.

```
def build(self):
    cmake = CMake(self)
    cmake.configure()
    cmake.build()
    cmake.install()
    # equivalent to
    # args += ['-DCMAKE_INSTALL_PREFIX="%s"' % self.package_folder]
    # self.run('cmake "%s/src" %s %s'
    #          % (self.source_folder, cmake.command_line, ' '.join( args ) ) )
    # self.run("cmake --build . --target install %s" % cmake.build_config)

def package(self):
    # nothing to do here now
```

The `package_info()` method is still necessary, as there is no possible way to automatically extract the information of the necessary libraries, defines and flags for different build configurations from the cmake install.

12.13 How to collaborate on other users' packages

If a certain existing package does not work for you, or you need to store pre-compiled binaries for a platform not provided by the original package creator, you might still be able to do so:

12.13.1 Collaborate from source repository

If the original package creator has the package recipe in a repository, this would be the simplest approach. Just clone the package recipe in your machine, change something if you want, and then export the package recipe under your own user name. Point your project's `[requires]` to the new package name, and use it as usual:

```
$ git clone <repository>
$ cd <repository>
//make changes if desired
$ conan export . <youruser/yourchannel>
```

You can just directly run:

```
$ conan create . demo/testing
```

Once you have generated the desired binaries, you can store your pre-compiled binaries in your bintray repository or in your own Conan server:

```
$ conan upload Package/0.1@myuser/stable -r=myremote --all
```

Finally, if you made useful changes, you might want to create a pull request to the original repository of the package creator.

12.13.2 Copy a package

If you don't need to modify the original package creator recipe, it is fine to just copy the package in your local storage. You can copy the recipes and existing binary packages. This could be enough for caching existing binary packages from the orginal remote into your own remote, under your own username:

```
$ conan copy Poco/1.7.8p3@pocoproject/stable myuser/testing
$ conan upload Poco/1.7.8p3@myuser/testing -r=myremote --all
```

12.14 How to link with Apple Frameworks

It is common in OSx that your conan package needs to link with a complete Apple framework, and, of course, you want to propagate this information to all projects/libraries that uses your package.

With regular libraries we use `self.cpp_info.libs` object to append to it all the libraries:

```
def package_info(self):

    self.cpp_info.libs = ["SDL2"]
    self.cpp_info.libs.append("OpenGL32")
```

With frameworks we need to declare the "-framework flag" as a linker flag:

```
def package_info(self):

    self.cpp_info.libs = ["SDL2"]

    self.cpp_info.exelinkflags.append("-framework Carbon")
    self.cpp_info.exelinkflags.append("-framework CoreAudio")
    self.cpp_info.exelinkflags.append("-framework Security")
    self.cpp_info.exelinkflags.append("-framework IOKit")

    self.cpp_info.sharedlinkflags = self.cpp_info.exelinkflags
```

In the previous example we are using `self.cpp_info.exelinkflags`. If we are using CMake to consume this package, it will only link those frameworks if we are building an executable and `sharedlinkflags` will only apply if we are building a shared library.

If we are not using CMake to consume this package `sharedlinkflags` and `exelinkflags` are used indistinctly. In the example above we are assigning in the last line `sharedlinkflags` with `exelinkflags`, so no matter what the consumer will build, it will indicate to the linker to link with the specified frameworks.

12.15 How to collect licenses of dependencies

With the `imports` feature it is possible to collect the License files from all packages in the dependency graph. Please note that the licenses are artifacts that must exist in the binary packages to be collected, as different binary packages

might have different licenses. E.g., A package creator might provide a different license for static or shared linkage with different "License" files if they want to.

Also, we will assume the convention that the package authors will provide a "License" (case not important) file at the root of their packages.

In *conanfile.txt* we would use the following syntax:

```
[imports]
., license* -> ./licenses @ folder=True, ignore_case=True
```

And in *conanfile.py* we will use the `imports()` method:

```
def imports(self):
     self.copy("license*", dst="licenses", folder=True, ignore_case=True)
```

In both cases, after **`conan install`**, it will store all the found License files inside the local **licenses** folder, wich will contain one subfolder per dependency with the license file inside.

12.16 How to capture package version from text or build files

It is common that a library version number would be already encoded in a text file, in some build scripts, etc. Lets take as an example that we have the following library layout, that we want to create a package from it:

```
conanfile.py
CMakeLists.txt
src
   hello.cpp
   ...
```

The *CMakeLists.txt* will have some variables to define the library version number. Lets assume for simplicity that it has some line like:

```
cmake_minimum_required(VERSION 2.8)
set(MY_LIBRARY_VERSION 1.2.3) # This is the version we want
add_library(hello src/hello.cpp)
```

We will typically have in our *conanfile.py* package recipe:

```
class HelloConan(ConanFile):
    name = "Hello"
    version = "1.2.3"
```

Usually this takes very little maintenance, and when the CMakeLists version is bumped, the *conanfile.py* version is bumped too. But if you want to only have to update the *CMakeLists.txt* version, you can extract the version dynamically, with:

```
from conans import ConanFile
from conans.tools import load
import re

def get_version():
    try:
        content = load("CMakeLists.txt")
        version = re.search(b"set\(MY_LIBRARY_VERSION (.*)\)", content).group(1)
        return version.strip()
```

(continues on next page)

(continued from previous page)

```
    except Exception as e:
        return None

class HelloConan(ConanFile):
    name = "Hello"
    version = get_version()
```

Even if the *CMakeLists.txt* file is not exported to the local cache, it will still work, as the `get_version()` function returns None when it is not found, then taking the version number from the package metadata (layout).

12.17 How to use Conan as other language package manager

Conan is a generic package manager. In the *getting started* section we saw how to use conan and manage a C/C++ library, like POCO.

But conan just provided some tools, related with C/C++ (like some generators and the cpp_info), to offer a better user experience. The general basis of Conan can be used with other programming languages.

Obviously, this does not try to compete with other package managers. Conan is a C and C++ package manager, focused on C and C++ developers. But when we realized that this was possible, we thought it was a good way to showcase its power, simplicity and versatility.

And of course, if you are doing C/C++ and occasionally you need some package from other language in your workflow, as in the conan package recipes themselves, or for some other tooling, you might find this functionality useful.

12.17.1 Conan: A Go package manager

The source code

You can just clone the following example repository:

```
$ git clone https://github.com/lasote/conan-goserver-example
```

Or, alternatively, manually create the folder and copy the following files inside:

```
$ mkdir conan-goserver-example
$ cd conan-goserver-example
$ mkdir src
$ mkdir src/server
```

The files are:

src/server/main.go is a small http server that will answer "Hello world!" if we connect to it.

```
package main

import "github.com/go-martini/martini"

func main() {
  m := martini.Classic()
  m.Get("/", func() string {
    return "Hello world!"
  })
```

(continues on next page)

(continued from previous page)

```
  m.Run()
}
```

Declaring and installing dependencies

Create a *conanfile.txt*, with the following content:

Listing 5: *conanfile.txt*

```
[requires]
go-martini/1.0@lasote/stable

[imports]
src, * -> ./deps/src
```

Our project requires a package, **go-martini/1.0@lasote/stable**, and we indicate that all **src contents** from all our requirements have to be copied to *./deps/src*.

The package go-martini depends on go-inject, so Conan will handle automatically the go-inject dependency.

```
$ conan install .
```

This command will download our packages and will copy the contents in the *./deps/src* folder.

Running our server

Just add the **deps** folder to `GOPATH`:

```
# Linux / Macos
$ export GOPATH=${GOPATH}:${PWD}/deps

# Windows
$ SET GOPATH=%GOPATH%;%CD%/deps
```

And run the server:

```
$ cd src/server
$ go run main.go
```

Open your browser and go to *localhost:9300*

```
Hello World!
```

Generating Go packages

Creating a Conan package for a Go library is very simple. In a Go project, you compile all the code from sources in the project itself, including all of its dependencies.

So we don't need to take care of settings at all. Architecture, compiler, operating system, etc. are only relevant for pre-compiled binaries. Source code packages are settings agnostic.

Let's take a look at the *conanfile.py* of the **go inject** library:

Listing 6: *conanfile.py*

```
from conans import ConanFile

class InjectConan(ConanFile):
    name = "go-inject"
    version = "1.0"

    def source(self):
        self.run("git clone https://github.com/codegangsta/inject.git")
        self.run("cd inject && git checkout v1.0-rc1")  # TAG v1.0-rc1

    def package(self):
        self.copy(pattern='*', dst='src/github.com/codegangsta/inject', src="inject",
↪ keep_path=True)
```

If you have read the *Building a hello world package*, the previous code may look quite simple to you.

We want to pack **version 1.0** of the **go inject** library, so the **version** variable is **"1.0"**. In the `source()` method, we declare how to obtain the source code of the library, in this case just by cloning the github repository and making a checkout of the **v1.0-rc1** tag. In the `package()` method, we are just copying all the sources to a folder named "src/github.com/codegangsta/inject".

This way, we can keep importing the library in the same way:

```
import "github.com/codegangsta/inject"
```

We can export and upload the package to a remote and we are done:

```
$ conan export . lasote/stable  # Or any other user/channel
$ conan upload go-inject/1.0@lasote/stable --all
```

Now look at the **go martini** conanfile:

Listing 7: *conanfile.py*

```
from conans import ConanFile

class InjectConan(ConanFile):
    name = "go-martini"
    version = "1.0"
    requires = 'go-inject/1.0@lasote/stable'

    def source(self):
        self.run("git clone https://github.com/go-martini/martini.git")
        self.run("cd martini && git checkout v1.0")  # TAG v1.0

    def package(self):
        self.copy(pattern='*', dst='src/github.com/go-martini/martini', src="martini
↪", keep_path=True)
```

It is very similar. The only difference is the `requires` variable. It defines the **go-inject/1.0@lasote/stable** library, as a requirement.

```
$ conan export . lasote/stable  # Or any other user/channel
$ conan upload go-martini/1.0@lasote/stable  --all
```

Now we are able to use them easily and without the problems of versioning with github checkouts.

12.17.2 Conan: A Python package manager

Conan is a C and C++ package manager, and to deal with the vast variability of C and C++ build systems, compilers, configurations, etc., it was designed with a great flexibility in mind, trying to let the users do almost what they want. This is one of the reasons to use Python as the scripting language for Conan package recipes.

With this flexibility, Conan is able to do very different tasks: package Visual Studio modules, *package Go code*, build packages from sources or from binaries retrieved from elsewhere, etc.

Python code can be reused and packaged with Conan to share functionalities or tools among *conanfile.py* files. Here we can see a full example of Conan as a Python package manager.

A full Python and C/C++ package manager

The real utility of this is that Conan is a C and C++ package manager. So if we want to create a python package that wraps the functionality of, lets say the Poco C++ library, it can be easily done. Poco itself has transitive (C/C++) dependencies, but they are already handled by Conan. Furthermore, a very interesting thing is that nothing has to be done in advance for that library, thanks to useful tools as **pybind11**, that allows to create python bindings easily.

So let's build a package with the following files:

- *conanfile.py*: The package recipe.
- *__init__.py*: necessary file, blank.
- *pypoco.cpp*: The C++ code with the `pybind11` wrapper for Poco that generates a python extension (a shared library that can be imported from python).
- *CMakeLists.txt*: cmake build file that is able to compile *pypoco.cpp* into a Python extension (*pypoco.pyd* in Windows, *pypoco.so* in Linux)
- *poco.py*: A Python file that makes use of the pypoco Python binary extension built with *pypoco.cpp*.
- *test_package/conanfile.py*: A test consumer recipe for convenience to create and test the package.

The *pypoco.cpp* file can be coded easily thanks to the elegant `pybind11` library:

Listing 8: pypoco.cpp

```
#include <pybind11/pybind11.h>
#include "Poco/Random.h"

using Poco::Random;
namespace py = pybind11;

PYBIND11_PLUGIN(pypoco) {
    py::module m("pypoco", "pybind11 example plugin");
    py::class_<Random>(m, "Random")
        .def(py::init<>())
        .def("nextFloat", &Random::nextFloat);
    return m.ptr();
}
```

And the *poco.py* file is straigthforward:

Listing 9: *poco.py*

```
import sys
import pypoco
```

(continues on next page)

(continued from previous page)

```
def random_float():
    r = pypoco.Random()
    return r.nextFloat()
```

The *conanfile.py* has a few more lines than the above, but it is still quite easy to understand:

Listing 10: *conanfile.py*

```
from conans import ConanFile, tools, CMake

class PocoPyReuseConan(ConanFile):
    name = "PocoPy"
    version = "0.1"
    requires = "Poco/1.9.0@pocoproject/stable", "pybind11/any@memsharded/stable"
    settings = "os", "compiler", "arch", "build_type"
    exports = "*"
    generators = "cmake"
    build_policy = "missing"

    def build(self):
        cmake = CMake(self)
        pythonpaths = "-DPYTHON_INCLUDE_DIR=C:/Python27/include -DPYTHON_LIBRARY=C:/
↪Python27/libs/python27.lib"
        self.run('cmake %s %s -DEXAMPLE_PYTHON_VERSION=2.7' % (cmake.command_line,␣
↪pythonpaths))
        self.run("cmake --build . %s" % cmake.build_config)

    def package(self):
        self.copy('*.py*')
        self.copy("*.so")

    def package_info(self):
        self.env_info.PYTHONPATH.append(self.package_folder)
```

The recipe now declares 2 `requires`, the **Poco library** and the **pybind11 library** that we will be using to create the binary extension.

As we are actually building some C++ code, we need a few important things:

- Input `settings` that define the OS, compiler, version and architecture we are using to build our extension. This is necessary because the binary we are building must match the architecture of the python interpreter that we will be using.
- The `build()` method is used actually to invoke CMake. See we had to hardcode the python path in the example, as the *CMakeLists.txt* call to `find_package(PythonLibs)` didn't find my python installed in *C:/Python27*, quite a standard path. I have added the `cmake` generator too to be able to easily use the declared `requires` build information inside my *CMakeLists.txt*.
- The *CMakeLists.txt* is not posted here, but is basically the one used in the pybind11 example with just 2 lines to include the conan generated cmake file for dependencies. It can be inspected in the github repo.
- Note that we are using Python 2.7 as an input option. If necessary, more options for other interpreters/architectures could be easily provided, as well as avoiding the hardcoded paths. Even the Python interpreter itself could be packaged in a conan package.

The above recipe will generate a different binary for different compilers or versions. As the binary is being wrapped by python, we could avoid this and use the same binary for different setups, modifying this behavior with the

`conan_info()` method.

```
$ conan export . memsharded/testing
$ conan install PocoPy/0.1@memsharded/testing -s arch=x86 -g virtualenv
$ activate
$ python
>>> import poco
>>> poco.random_float()
0.697845458984375
```

Now the first invocation of **`conan install`** will build retrieve the dependencies and build the package. The next invocation will use the cached binaries and be much faster. Note how we have to specify `-s arch=x86` to build matching the architecture of the python interpreter to be used, in our case, 32 bits.

We can also read in the output of the **`conan install`** the dependencies that are being pulled:

```
Requirements
    OpenSSL/1.0.2l@conan/stable from conan.io
    Poco/1.9.0@pocoproject/stable from conan.io
    PocoPy/0.1@memsharded/testing from local
    pybind11/any@memsharded/stable from conan.io
    zlib/1.2.11@conan/stable from conan.io
```

This is the great thing about using Conan for this task, by depending on Poco, other C and C++ transitive dependencies are being retrieved and used in the application.

If you want to have a further look to the code of these examples, you can check this github repo. The above examples and code have been tested only in Win10, VS14u2, but might work with other configurations with little or no extra work.

12.18 How to manage SSL (TLS) certificates

12.18.1 Server certificate validation

By default, when a remote is added, if the URL schema is `https`, the Conan client will verify the certificate using a list of authorities declared in the `cacert.pem` file located in the conan home (~/.conan).

If you have a self signed certificate (not signed by any authority) you have two options:

- Use the *conan remote* command to disable the SSL verification.
- Append your server `crt` file to the `cacert.pem` file.

12.18.2 Client certificates

If your server is requiring client certificates to validate a connection from a Conan client, you need to create two files in the conan home directory (default ~/.conan):

- A file `client.crt` with the client certificate.
- A file `client.key` with the private key.

Note: You can create only the `client.crt` file containing both the certificate and the private key concatenated and not create the `client.key`

If you are a familiar with the curl tool, this mechanism is similar to specify the `--cert` / `--key` parameters.

CHAPTER

THIRTEEN

REFERENCE

General information about the commands, configuration files, etc.

Contents:

13.1 Commands

13.1.1 Consumer commands

Commands related with the installation and usage of Conan packages:

conan install

```
$ conan install [-h] [-g GENERATOR] [-if INSTALL_FOLDER] [-m [MANIFESTS]]
                [-mi [MANIFESTS_INTERACTIVE]] [-v [VERIFY]]
                [--no-imports] [-j JSON] [-b [BUILD]] [-e ENV]
                [-o OPTIONS] [-pr PROFILE] [-r REMOTE] [-s SETTINGS] [-u]
                path_or_reference
```

Installs the requirements specified in a recipe (conanfile.py or conanfile.txt). It can also be used to install a concrete package specifying a reference. If any requirement is not found in the local cache, it will retrieve the recipe from a remote, looking for it sequentially in the configured remotes. When the recipes have been downloaded it will try to download a binary package matching the specified settings, only from the remote from which the recipe was retrieved. If no binary package is found, it can be build from sources using the '–build' option. When the package is installed, Conan will write the files for the specified generators.

```
positional arguments:
  path_or_reference     Path to a folder containing a recipe (conanfile.py or
                        conanfile.txt) or to a recipe file. e.g.,
                        ./my_project/conanfile.txt. It could also be a
                        reference

optional arguments:
  -h, --help            show this help message and exit
  -g GENERATOR, --generator GENERATOR
                        Generators to use
  -if INSTALL_FOLDER, --install-folder INSTALL_FOLDER
                        Use this directory as the directory where to put the
                        generatorfiles. e.g., conaninfo/conanbuildinfo.txt
  -m [MANIFESTS], --manifests [MANIFESTS]
                        Install dependencies manifests in folder for later
```

(continues on next page)

(continued from previous page)

```
                      verify. Default folder is .conan_manifests, but can be
                      changed
-mi [MANIFESTS_INTERACTIVE], --manifests-interactive [MANIFESTS_INTERACTIVE]
                      Install dependencies manifests in folder for later
                      verify, asking user for confirmation. Default folder
                      is .conan_manifests, but can be changed
-v [VERIFY], --verify [VERIFY]
                      Verify dependencies manifests against stored ones
--no-imports          Install specified packages but avoid running imports
-j JSON, --json JSON  Path to a json file where the install information will
                      be written
-b [BUILD], --build [BUILD]
                      Optional, use it to choose if you want to build from
                      sources: --build Build all from sources, do not use
                      binary packages. --build=never Never build, use binary
                      packages or fail if a binary package is not found.
                      --build=missing Build from code if a binary package is
                      not found. --build=outdated Build from code if the
                      binary is not built with the current recipe or when
                      missing binary package. --build=[pattern] Build always
                      these packages from source, but never build the
                      others. Allows multiple --build parameters. 'pattern'
                      is a fnmatch file pattern of a package name. Default
                      behavior: If you don't specify anything, it will be
                      similar to '--build=never', but package recipes can
                      override it with their 'build_policy' attribute in the
                      conanfile.py.
-e ENV, --env ENV     Environment variables that will be set during the
                      package build, -e CXX=/usr/bin/clang++
-o OPTIONS, --options OPTIONS
                      Define options values, e.g., -o Pkg:with_qt=true
-pr PROFILE, --profile PROFILE
                      Apply the specified profile to the install command
-r REMOTE, --remote REMOTE
                      Look in the specified remote server
-s SETTINGS, --settings SETTINGS
                      Settings to build the package, overwriting the
                      defaults. e.g., -s compiler=gcc
-u, --update          Check updates exist from upstream remotes
```

conan install executes methods of a *conanfile.py* in the following order:

1. `config_options()`
2. `configure()`
3. `requirements()`
4. `package_id()`
5. `package_info()`
6. `deploy()`

Note this describes the process of installing a pre-built binary package. If the package has to be built, **conan install --build** executes the following:

1. `config_options()`
2. `configure()`

3. `requirements()`
4. `package_id()`
5. `build_requirements()`
6. `build_id()`
7. `system_requirements()`
8. `source()`
9. `imports()`
10. `build()`
11. `package()`
12. `package_info()`
13. `deploy()`

Examples

- Install a package requirement from a `conanfile.txt`, saved in your current directory with one option and setting (other settings will be defaulted as defined in `<userhome>/.conan/profiles/default`):

```
$ conan install . -o PkgName:use_debug_mode=on -s compiler=clang
```

Note: You have to take into account that **settings** are cached as defaults in the **conaninfo.txt** file, so you don't have to type them again and again in the **conan install** or **conan create** commands.

However, the default **options** are defined in your **conanfile**. If you want to change the default options across all your **conan install** commands, change them in the **conanfile**. When you change the **options** on the command line, they are only changed for one shot. Next time, **conan install** will take the **conanfile** options as default values, if you don't specify them again in the command line.

- Install the **OpenCV/2.4.10@lasote/testing** reference with its default options and default settings from `<userhome>/.conan/profiles/default`:

```
$ conan install opencv/2.4.10@lasote/testing
```

- Install the **OpenCV/2.4.10@lasote/testing** reference updating the recipe and the binary package if new upstream versions are available:

```
$ conan install opencv/2.4.10@lasote/testing --update
```

build options

Both the conan **install** and **create** commands have options to specify whether conan should try to build things or not:

- **`--build=never`**: This is the default option. It is not necessary to write it explicitly. Conan will not try to build packages when the requested configuration does not match, in which case it will throw an error.
- **`--build=missing`**: Conan will try to build from source, all packages of which the requested configuration was not found on any of the active remotes.
- **`--build=outdated`**: Conan will try to build from code if the binary is not built with the current recipe or when missing binary package.

- **--build=[pattern]**: A fnmatch file pattern of a package name. e.j `zl*` will match `zlib` package. Conan will force the build of the packages, the name of which matches the given **pattern**. Several patterns can be specified, chaining multiple options, e.g. **--build=pattern1 --build=pattern2**.
- **--build**: Always build everything from source. Produces a clean re-build of all packages and transitively dependent packages

env variables

With the **-e** parameters you can define:

- Global environment variables (**-e SOME_VAR="SOME_VALUE"**). These variables will be defined before the *build* step in all the packages and will be cleaned after the *build* execution.
- Specific package environment variables (**-e zlib:SOME_VAR="SOME_VALUE"**). These variables will be defined only in the specified packages (e.g. zlib).

You can specify this variables not only for your direct `requires` but for any package in the dependency graph.

If you want to define an environment variable but you want to append the variables declared in your requirements you can use the [] syntax:

```
$ conan install . -e PYTHONPATH=[/other/path]
```

This way the first entry in the PYTHONPATH variable will be **/other/path** but the PYTHONPATH values declared in the requirements of the project will be appended at the end using the system path separator.

settings

With the **-s** parameters you can define:

- Global settings (**-s compiler="Visual Studio"**). Will apply to all the requires.
- Specific package settings (**-s zlib:compiler="MinGW"**). Those settings will be applied only to the specified packages.

You can specify custom settings not only for your direct `requires` but for any package in the dependency graph.

options

With the **-o** parameters you can only define specific package options.

```
$ conan install . -o zlib:shared=True
$ conan install . -o zlib:shared=True -o bzip2:option=132
# you can also apply the same options to many packages with wildcards:
$ conan install . -o *:shared=True
```

Note: You can use *profiles* files to create predefined sets of **settings**, **options** and **environment variables**.

conan config

```
$ conan config [-h] {rm,set,get,install} ...
```

Manages Conan configuration. Edits the conan.conf or installs config files.

```
positional arguments:
  {rm,set,get,install}  sub-command help
    rm                  Remove an existing config element
    set                 Set a value for a configuration item
    get                 Get the value of configuration item
    install             install a full configuration from a local or remote
                        zip file

optional arguments:
  -h, --help            show this help message and exit
```

Examples

- Change the logging level to 10:

```
$ conan config set log.level=10
```

- Get the logging level:

```
$ conan config get log.level
$> 10
```

conan config install

The `config install` is intended to share the Conan client configuration. For example, in a company or organization, is important to have common `settings.yml`, `profiles`, etc.

It retrieves a zip file from a local directory or url and apply the files in the local Conan configuration.

The zip can contain only a subset of all the allowed configuration files, only the present files will be replaced, except the **conan.conf** file, that will apply only the declared variables in the zipped `conan.conf` file and will keep the rest of the local variables.

The **profiles files**, that will be overwritten if already present, but won't delete any other profile file that the user has in the local machine.

All files in the zip will be copied to the conan home directory. These are the special files and the rules applied to merge them:

File	How it is applied
profiles/MyProfile	Overrides the local ~/.conan/profiles/MyProfile if already exists
settings.yml	Overrides the local ~/.conan/settings.yml
remotes.txt	Overrides remotes. Will remove remotes that are not present in file
config/conan.conf	Merges the variables, overriding only the declared variables

The local cache *registry.txt* file contains the remotes definitions, as well as the mapping from packages to remotes. In general it is not a good idea to add it to the installed files.

The specified URL will be stored in the `general.config_install` variable of the `conan.conf` file, so following calls to **`conan config install`** command doesn't need to specify the URL.

Examples:

- Install the configuration from an url:

```
$ conan config install http://url/to/some/config.zip
```

Conan config command stores the specified URL in the conan.conf `general.config_install` variable.

- Install from an url skipping SSL verification:

```
$ conan config install http://url/to/some/config.zip --verify-ssl=False
```

This will disable the SSL check of the certificate. This option is defaulted to `True`.

- Refresh the configuration again:

```
$ conan config install
```

It's not needed to specify the url again, it is already stored.

- Install the configuration from a local path:

```
$ conan config install /path/to/some/config.zip
```

conan get

```
$ conan get [-h] [-p PACKAGE] [-r REMOTE] [-raw] reference [path]
```

Gets a file or list a directory of a given reference or package.

```
positional arguments:
  reference             package recipe reference
  path                  Path to the file or directory. If not specified will
                        get the conanfile if only a reference is specified and
                        a conaninfo.txt file contents if the package is also
                        specified

optional arguments:
  -h, --help            show this help message and exit
  -p PACKAGE, --package PACKAGE
                        Package ID
  -r REMOTE, --remote REMOTE
                        Get from this specific remote
  -raw, --raw           Do not decorate the text
```

Examples:

- Print the conanfile.py from a remote package:

```
$ conan get zlib/1.2.8@conan/stable -r conan-center
```

- List the files for a local package recipe:

```
$ conan get zlib/1.2.11@conan/stable .

Listing directory '.':
 CMakeLists.txt
 conanfile.py
 conanmanifest.txt
```

- Print a file from a recipe folder:

```
$ conan get zlib/1.2.11@conan/stable conanmanifest.txt
```

- Print the conaninfo.txt file for a binary package:

```
$ conan get zlib/1.2.11@conan/stable -p 09512ff863f37e98ed748eadd9c6df3e4ea424a8
```

```
[settings]
    arch=x86_64
    build_type=Release
    compiler=apple-clang
    compiler.version=8.1
    os=Macos

[requires]


[options]
    # ...
```

- List the files from a binary package in a remote:

```
$ conan get zlib/1.2.11@conan/stable . -p␣
↪09512ff863f37e98ed748eadd9c6df3e4ea424a8 -r conan-center

Listing directory '.':
 conan_package.tgz
 conaninfo.txt
 conanmanifest.txt
```

conan info

```
$ conan info [-h] [--paths] [-bo BUILD_ORDER] [-g GRAPH]
             [-if INSTALL_FOLDER] [-j [JSON]] [-n ONLY]
             [--package-filter [PACKAGE_FILTER]] [-b [BUILD]] [-e ENV]
             [-o OPTIONS] [-pr PROFILE] [-r REMOTE] [-s SETTINGS] [-u]
             path_or_reference
```

Gets information about the dependency graph of a recipe. It can be used with a recipe or a reference for any existing package in your local cache.

```
positional arguments:
  path_or_reference     Path to a folder containing a recipe (conanfile.py or
                        conanfile.txt) or to a recipe file. e.g.,
                        ./my_project/conanfile.txt. It could also be a
                        reference

optional arguments:
  -h, --help            show this help message and exit
  --paths               Show package paths in local cache
  -bo BUILD_ORDER, --build-order BUILD_ORDER
                        given a modified reference, return an ordered list to
                        build (CI)
  -g GRAPH, --graph GRAPH
                        Creates file with project dependencies graph. It will
                        generate a DOT or HTML file depending on the filename
```

(continues on next page)

(continued from previous page)

```
                        extension
  -if INSTALL_FOLDER, --install-folder INSTALL_FOLDER
                        local folder containing the conaninfo.txt and
                        conanbuildinfo.txt files (from a previous conan
                        install execution). Defaulted to current folder,
                        unless --profile, -s or -o is specified. If you
                        specify both install-folder and any setting/option it
                        will raise an error.
  -j [JSON], --json [JSON]
                        Only with --build_order option, return the information
                        in a json. e.j --json=/path/to/filename.json or --json
                        to output the json
  -n ONLY, --only ONLY  Show only the specified fields: "id", "build_id",
                        "remote", "url", "license", "requires", "update",
                        "required", "date", "author", "None". '--paths'
                        information can also be filtered with options
                        "export_folder", "build_folder", "package_folder",
                        "source_folder". Use '--only None' to show only
                        references.
  --package-filter [PACKAGE_FILTER]
                        Print information only for packages that match the
                        filter pattern e.g., MyPackage/1.2@user/channel or
                        MyPackage*
  -b [BUILD], --build [BUILD]
                        Given a build policy, return an ordered list of
                        packages that would be built from sources during the
                        install command
  -e ENV, --env ENV     Environment variables that will be set during the
                        package build, -e CXX=/usr/bin/clang++
  -o OPTIONS, --options OPTIONS
                        Define options values, e.g., -o Pkg:with_qt=true
  -pr PROFILE, --profile PROFILE
                        Apply the specified profile to the install command
  -r REMOTE, --remote REMOTE
                        Look in the specified remote server
  -s SETTINGS, --settings SETTINGS
                        Settings to build the package, overwriting the
                        defaults. e.g., -s compiler=gcc
  -u, --update          Check updates exist from upstream remotes
```

Examples:

```
$ conan info .
$ conan info myproject_folder
$ conan info myproject_folder/conanfile.py
$ conan info Hello/1.0@user/channel
```

The output will look like:

```
Dependency/0.1@user/channel
 ID: 5ab84d6acfe1f23c4fae0ab88f26e3a396351ac9
 BuildID: None
 Remote: None
 URL: http://...
 License: MIT
 Updates: Version not checked
 Creation date: 2017-10-31 14:45:34
```

(continues on next page)

(continued from previous page)

```
 Required by:
    Hello/1.0@user/channel

Hello/1.0@user/channel
 ID: 5ab84d6acfe1f23c4fa5ab84d6acfe1f23c4fa8
 BuildID: None
 Remote: None
 URL: http://...
 License: MIT
 Updates: Version not checked
 Required by:
    Project
 Requires:
    Hello0/0.1@user/channel
```

conan info builds the complete dependency graph, like **conan install** does. The main difference is that it doesn't try to install or build the binaries, but the package recipes will be retrieved from remotes if necessary.

It is very important to note, that the **info** command outputs the dependency graph for a given configuration (settings, options), as the dependency graph can be different for different configurations. Then, the input to the **conan info** commmand is the same as **conan install**, the configuration can be specified directly with settings and options, or using profiles.

Also, if you did a previous **conan install** with a specific configuration, or maybe different installs with different configurations, you can reuse that information with the **--install-folder** argument:

```
$ # dir with a conanfile.txt
$ mkdir build_release && cd build_release
$ conan install .. --profile=gcc54release
$ cd .. && mkdir build_debug && cd build_debug
$ conan install .. --profile=gcc54debug
$ cd ..
$ conan info . --install-folder=build_release
> info for the release dependency graph install
$ conan info . --install-folder=build_debug
> info for the debug dependency graph install
```

It is possible to use the **conan info** command to extract useful information for Continuous Integration systems. More precisely, it has the **--build_order, -bo** option, that will produce a machine-readable output with an ordered list of package references, in the order they should be built. E.g., lets assume that we have a project that depends on Boost and Poco, which in turn depends on OpenSSL and ZLib transitively. So we can query our project with a reference that has changed (most likely due to a git push on that package):

```
$ conan info . -bo zlib/1.2.11@conan/stable
[zlib/1.2.11@conan/stable], [OpenSSL/1.0.2l@conan/stable], [Boost/1.60.0@lasote/
↪stable, Poco/1.7.8p3@pocoproject/stable]
```

Note the result is a list of lists. When there is more than one element in one of the lists, it means that they are decoupled projects and they can be built in parallel by the CI system.

You can also specify the `ALL` argument, if you want just to compute the whole dependency graph build order

```
$ conan info . --build_order=ALL
> [zlib/1.2.11@conan/stable], [OpenSSL/1.0.2l@conan/stable], [Boost/1.60.0@lasote/
↪stable, Poco/1.7.8p3@pocoproject/stable]
```

Also you can get a list of nodes that would be built (simulation) in an install command specifying a build policy with

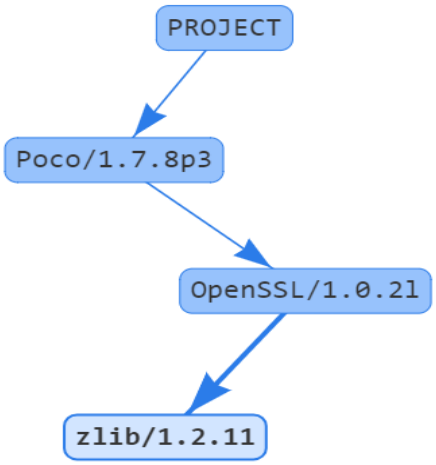
the `--build` parameter:

e.g., If I try to install `Boost/1.60.0@lasote/stable` recipe with `--build missing` build policy and `arch=x86`, which libraries will be built?

```
$ conan info Boost/1.60.0@lasote/stable --build missing -s arch=x86
bzip2/1.0.6@lasote/stable, zlib/1.2.8@lasote/stable, Boost/1.60.0@lasote/stable
```

You can generate a graph of your dependencies, in dot or html formats:

```
$ conan info .. --graph=file.html
$ file.html # or open the file, double-click
```



conan search

```
$ conan search [-h] [-o] [-q QUERY] [-r REMOTE] [--case-sensitive]
               [--raw] [--table TABLE]
               [pattern_or_reference]
```

Searches package recipes and binaries in the local cache or in a remote. If you provide a pattern, then it will search for existing package recipes matching it. If a full reference is provided (pkg/0.1@user/channel) then the existing binary packages for that reference will be displayed. If no remote is specified, the serach will be done in the local cache. Search is case sensitive, exact case has to be used. For case insensitive file systems, like Windows, case sensitive search can be forced with '–case-sensitive'.

```
positional arguments:
  pattern_or_reference  Pattern or package recipe reference, e.g., 'boost/*',
                        'MyPackage/1.2@user/channel'

optional arguments:
  -h, --help            show this help message and exit
  -o, --outdated        Show only outdated from recipe packages
  -q QUERY, --query QUERY
                        Packages query: 'os=Windows AND (arch=x86 OR
                        compiler=gcc)'. The 'pattern_or_reference' parameter
                        has to be a reference: MyPackage/1.2@user/channel
  -r REMOTE, --remote REMOTE
                        Remote to search in. '-r all' searches all remotes
```

(continues on next page)

(continued from previous page)

```
  --case-sensitive      Make a case-sensitive search. Use it to guarantee
                        case-sensitive search in Windows or other case-
                        insensitive file systems
  --raw                 Print just the list of recipes
  --table TABLE         Outputs html file with a table of binaries. Only valid
                        for a reference search
```

Examples

```
$ conan search zlib/*
$ conan search zlib/* -r=conan-center
```

To search for recipes in all defined remotes use `--all` (this is only valid for searching recipes, not binaries):

```
$ conan search zlib/* -r=all
```

If you use instead the full package recipe reference, you can explore the binaries existing for that recipe, also in a remote or in the local conan cache:

```
$ conan search Boost/1.60.0@lasote/stable
```

A query syntax is allowed to look for specific binaries, you can use `AND` and `OR` operators and parenthesis, with settings and also options.

```
$ conan search Boost/1.60.0@lasote/stable -q arch=x86_64
$ conan search Boost/1.60.0@lasote/stable -q "(arch=x86_64 OR arch=ARM) AND (build_
↪type=Release OR os=Windows)"
```

If you specify a query filter for a setting and the package recipe is not restricted by this setting, will find all packages:

```
class MyRecipe(ConanFile):
    settings="arch"
```

```
$ conan search MyRecipe/1.0@lasote/stable -q os=Windows
```

The query above will find all the `MyRecipe` binary packages, because the recipe doesn't declare "os" as a setting.

You can generate a table for all binaries from a given recipe with the `--table` option:

```
$ conan search zlib/1.2.11@conan/stable --table=file.html -r=conan-center
$ file.html # or open the file, double-click
```

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

13.1.2 Creator commands

Commands related to the creation of Conan recipes and packages:

conan create

```
$ conan create [-h] [-j JSON] [-k] [-kb] [-ne] [-tbf TEST_BUILD_FOLDER]
               [-tf TEST_FOLDER] [-m [MANIFESTS]]
               [-mi [MANIFESTS_INTERACTIVE]] [-v [VERIFY]] [-b [BUILD]]
               [-e ENV] [-o OPTIONS] [-pr PROFILE] [-r REMOTE]
               [-s SETTINGS] [-u]
               path reference
```

Builds a binary package for a recipe (conanfile.py). Uses the specified configuration in a profile or in -s settings, -o options etc. If a 'test_package' folder (the name can be configured with -tf) is found, the command will run the consumer project to ensure that the package has been created correctly. Check 'conan test' command to know more about 'test_folder' project.

```
positional arguments:
  path                  Path to a folder containing a conanfile.py or to a
```

(continues on next page)

(continued from previous page)

```
                        recipe file e.g., my_folder/conanfile.py
  reference             user/channel or pkg/version@user/channel (if name and
                        version not declared in conanfile.py) where the
                        pacakage will be created

optional arguments:
  -h, --help            show this help message and exit
  -j JSON, --json JSON  json file path where the install information will be
                        written to
  -k, -ks, --keep-source
                        Do not remove the source folder in local cache. Use
                        this for testing purposes only
  -kb, --keep-build     Do not remove the build folder in local cache. Use
                        this for testing purposes only
  -ne, --not-export     Do not export the conanfile.py
  -tbf TEST_BUILD_FOLDER, --test-build-folder TEST_BUILD_FOLDER
                        Working directory for the build of the test project.
  -tf TEST_FOLDER, --test-folder TEST_FOLDER
                        Alternative test folder name. By default it is
                        "test_package". Use "None" to skip the test stage
  -m [MANIFESTS], --manifests [MANIFESTS]
                        Install dependencies manifests in folder for later
                        verify. Default folder is .conan_manifests, but can be
                        changed
  -mi [MANIFESTS_INTERACTIVE], --manifests-interactive [MANIFESTS_INTERACTIVE]
                        Install dependencies manifests in folder for later
                        verify, asking user for confirmation. Default folder
                        is .conan_manifests, but can be changed
  -v [VERIFY], --verify [VERIFY]
                        Verify dependencies manifests against stored ones
  -b [BUILD], --build [BUILD]
                        Optional, use it to choose if you want to build from
                        sources: --build Build all from sources, do not use
                        binary packages. --build=never Never build, use binary
                        packages or fail if a binary package is not found.
                        --build=missing Build from code if a binary package is
                        not found. --build=outdated Build from code if the
                        binary is not built with the current recipe or when
                        missing binary package. --build=[pattern] Build always
                        these packages from source, but never build the
                        others. Allows multiple --build parameters. 'pattern'
                        is a fnmatch file pattern of a package name. Default
                        behavior: If you don't specify anything, it will be
                        similar to '--build=never', but package recipes can
                        override it with their 'build_policy' attribute in the
                        conanfile.py.
  -e ENV, --env ENV     Environment variables that will be set during the
                        package build, -e CXX=/usr/bin/clang++
  -o OPTIONS, --options OPTIONS
                        Define options values, e.g., -o Pkg:with_qt=true
  -pr PROFILE, --profile PROFILE
                        Apply the specified profile to the install command
  -r REMOTE, --remote REMOTE
                        Look in the specified remote server
  -s SETTINGS, --settings SETTINGS
                        Settings to build the package, overwriting the
                        defaults. e.g., -s compiler=gcc
```

(continues on next page)

(continued from previous page)

```
  -u, --update          Check updates exist from upstream remotes
```

This is the recommended way to create packages.

conan create . demo/testing is equivalent to:

```
$ conan export . demo/testing
$ conan install Hello/0.1@demo/testing --build=Hello
# package is created now, use test to test it
$ cd test_package
$ conan test . Hello/0.1@demo/testing
```

Tip: Sometimes you need to **skip/disable test stage** to avoid a failure while creating the package, i.e: when you are cross compiling libraries and target code cannot be executed in current host platform. In that case you can skip/disable the test package stage:

```
$ conan create . demo/testing --test-folder=None
```

conan create executes methods of a *conanfile.py* in the following order:

1. `config_options()`
2. `configure()`
3. `requirements()`
4. `package_id()`
5. `build_requirements()`
6. `build_id()`
7. `system_requirements()`
8. `source()`
9. `imports()`
10. `build()`
11. `package()`
12. `package_info()`

In case of installing a pre-built binary, steps from 5 to 11 will be skipped. Note that `deploy()` method is only used in **conan install**.

conan export

```
$ conan export [-h] [-k] path reference
```

Copies the recipe (conanfile.py & associated files) to your local cache. Use the 'reference' param to specify a user and channel where to export it. Once the recipe is in the local cache it can be shared, reused and to any remote with the 'conan upload' command.

```
positional arguments:
  path                  Path to a folder containing a conanfile.py or to a
                        recipe file e.g., my_folder/conanfile.py
  reference             user/channel, or Pkg/version@user/channel (if name and
                        version are not declared in the conanfile.py

optional arguments:
  -h, --help            show this help message and exit
  -k, -ks, --keep-source
                        Do not remove the source folder in local cache. Use
                        this for testing purposes only
```

The `export` command will run a linting of the package recipe, looking for possible inconsistencies, bugs and py2-3 incompatibilities. It is possible to customize the rules for this linting, as well as totally disabling it. Look at the `recipe_linter` and `pylintrc` variables in *conan.conf* and the `PYLINTRC` environment variable.

Examples

- Export a recipe using a full reference. Only valid if `name` and `version` are not declared in the recipe:

```
$ conan export . mylib/1.0@myuser/channel
```

- Export a recipe from any folder directory, under the `myuser/stable` user and channel:

```
$ conan export ./folder_name myuser/stable
```

- Export a recipe without removing the source folder in the local cache:

```
$ conan export . fenix/stable -k
```

conan export-pkg

```
$ conan export-pkg [-h] [-bf BUILD_FOLDER] [-e ENV] [-f]
                   [-if INSTALL_FOLDER] [-o OPTIONS] [-pr PROFILE]
                   [-pf PACKAGE_FOLDER] [-s SETTINGS] [-sf SOURCE_FOLDER]
                   path reference
```

Exports a recipe & creates a package with given files calling the package() method applied to the local folders '–source-folder' and '–build-folder' and creates a new package in the local cache for the specified 'reference' and for the specified '–settings', '–options' and or '–profile'.

```
positional arguments:
  path                  Path to a folder containing a conanfile.py or to a
                        recipe file e.g., my_folder/conanfile.py
  reference             user/channel or pkg/version@user/channel (if name and
                        version are not declared in the conanfile.py)

optional arguments:
  -h, --help            show this help message and exit
  -bf BUILD_FOLDER, --build-folder BUILD_FOLDER
                        Directory for the build process. Defaulted to the
                        current directory. A relative path to current
                        directory can also be specified
  -e ENV, --env ENV     Environment variables that will be set during the
                        package build, -e CXX=/usr/bin/clang++
  -f, --force           Overwrite existing package if existing
```

(continues on next page)

(continued from previous page)

```
  -if INSTALL_FOLDER, --install-folder INSTALL_FOLDER
                        Directory containing the conaninfo.txt and
                        conanbuildinfo.txt files (from previous 'conan
                        install'). Defaulted to --build-folder If these files
                        are found in the specified folder and any of '-e',
                        '-o', '-pr' or '-s' arguments are used, it will raise
                        an error.
  -o OPTIONS, --options OPTIONS
                        Define options values, e.g., -o pkg:with_qt=true
  -pr PROFILE, --profile PROFILE
                        Profile for this package
  -pf PACKAGE_FOLDER, --package-folder PACKAGE_FOLDER
                        folder containing a locally created package. If a
                        value is given, it won't call the recipe 'package()'
                        method, and will run a copy of the provided folder.
  -s SETTINGS, --settings SETTINGS
                        Define settings values, e.g., -s compiler=gcc
  -sf SOURCE_FOLDER, --source-folder SOURCE_FOLDER
                        Directory containing the sources. Defaulted to the
                        conanfile's directory. A relative path to current
                        directory can also be specified
```

conan export-pkg executes the following methods of a *conanfile.py* whenever `--package-folder` is used:

1. `config_options()`
2. `configure()`
3. `requirements()`
4. `package_id()`

In case a package folder is not specified, this command will also execute:

5. `package()`

Note that this is **not** the normal or recommended flow for creating Conan packages, as packages created this way will not have a reproducible build from sources. This command should be used when:

- It is not possible to build the packages from sources (only pre-built binaries available).
- You are developing your package locally and want to export the built artifacts to the local cache.

The command **conan new <ref> --bare** will create a simple recipe that could be used in combination with the `export-pkg` command. Check this *How to package existing binaries*.

export-pkg has two different modes of operation:

- Specifying **--package-folder** will perform a copy of the given folder, without executing the `package()` method. Use it if you have already created the package, for example with **conan package** or with `cmake.install()` from the `build()` step.
- Specifying **--build-folder** and/or **--source-folder** will execute the `package()` method, to filter, select and arrange the layout of the artifacts.

Examples:

- Create a package from a directory containing the binaries for Windows/x86/Release:

  Having these files:

```
Release_x86/lib/libmycoollib.a
Release_x86/lib/other.a
Release_x86/include/mylib.h
Release_x86/include/other.h
```

Run:

```
$ conan new Hello/0.1 --bare  # In case you still don't have a recipe for the␣
↪binaries
$ conan export-pkg . Hello/0.1@user/stable -s os=Windows -s arch=x86 -s build_
↪type=Release --build-folder=Release_x86
```

- Create a package from a user folder build and sources folders:

  Given these files in the current folder

```
sources/include/mylib.h
sources/src/file.cpp
build/lib/mylib.lib
build/lib/mylib.tmp
build/file.obj
```

  And assuming the `Hello/0.1@user/stable` recipe has a `package()` method like this:

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

  Then, the following code will create a package in the conan local cache:

```
$ conan export-pkg . Hello/0.1@user/stable -pr=myprofile --source-folder=sources -
↪-build-folder=build
```

  And such package will contain just the files:

```
include/mylib.h
lib/mylib.lib
```

- Building a conan package (for architecture x86) in a local directory and then send it to the local cache:

  conanfile.py

```
from conans import ConanFile, CMake, tools

class LibConan(ConanFile):
    name = "Hello"
    version = "0.1"
    ...

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

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

    def package(self):
```

(continues on next page)

(continued from previous page)

```
        self.copy("*.h", dst="include", src="include")
        self.copy("*.lib", dst="lib", keep_path=False)
```

First we will call **conan source** to get our source code in the *src* directory, then **conan install** to install the requirements and generate the info files, **conan build** to build the package, and finally **conan export-pkg** to send the binary files to a package in the local cache:

```
$ conan source . --source-folder src
$ conan install --install-folder build_x86 -s arch=x86
$ conan build . --build-folder build_x86 --source-folder src
$ conan export-pkg . Hello/0.1@user/stable --build-folder build_x86
```

In this case, in the **conan export-pkg**, you don't need to specify the **-s arch=x86** or any other setting, option, or profile, because it will all the information in the **--build-folder** the *conaninfo.txt* and *conanbuildinfo.txt*' that have been created with **conan install**.

conan new

```
$ conan new [-h] [-t] [-i] [-c] [-s] [-b] [-cis] [-cilg] [-cilc] [-cio]
            [-ciw] [-ciglg] [-ciglc] [-ciccg] [-ciccc] [-cicco] [-gi]
            [-ciu CI_UPLOAD_URL]
            name
```

Creates a new package recipe template with a 'conanfile.py' and optionally, 'test_package' testing files.

```
positional arguments:
  name                  Package name, e.g.: "Poco/1.7.3" or complete reference
                        for CI scripts: "Poco/1.7.3@conan/stable"

optional arguments:
  -h, --help            show this help message and exit
  -t, --test            Create test_package skeleton to test package
  -i, --header          Create a headers only package template
  -c, --pure-c          Create a C language package only package, deleting
                        "self.settings.compiler.libcxx" setting in the
                        configure method
  -s, --sources         Create a package with embedded sources in "src"
                        folder, using "exports_sources" instead of retrieving
                        external code with the "source()" method
  -b, --bare            Create the minimum package recipe, without build()
                        methodUseful in combination with "export-pkg" command
  -cis, --ci-shared     Package will have a "shared" option to be used in CI
  -cilg, --ci-travis-gcc
                        Generate travis-ci files for linux gcc
  -cilc, --ci-travis-clang
                        Generate travis-ci files for linux clang
  -cio, --ci-travis-osx
                        Generate travis-ci files for OSX apple-clang
  -ciw, --ci-appveyor-win
                        Generate appveyor files for Appveyor Visual Studio
  -ciglg, --ci-gitlab-gcc
                        Generate GitLab files for linux gcc
  -ciglc, --ci-gitlab-clang
                        Generate GitLab files for linux clang
  -ciccg, --ci-circleci-gcc
```

(continues on next page)

(continued from previous page)

```
                        Generate CicleCI files for linux gcc
  -ciccc, --ci-circleci-clang
                        Generate CicleCI files for linux clang
  -cicco, --ci-circleci-osx
                        Generate CicleCI files for OSX apple-clang
  -gi, --gitignore      Generate a .gitignore with the known patterns to
                        excluded
  -ciu CI_UPLOAD_URL, --ci-upload-url CI_UPLOAD_URL
                        Define URL of the repository to upload
```

Examples:

- Create a new `conanfile.py` for a new package **mypackage/1.0@myuser/stable**

```
$ conan new mypackage/1.0
```

- Create also a `test_package` folder skeleton:

```
$ conan new mypackage/1.0 -t
```

- Create files for travis (both Linux and OSX) and appveyor Continuous Integration:

```
$ conan new mypackage/1.0@myuser/stable -t -cilg -cio -ciw
```

- Create files for gitlab (linux) Continuous integration and set upload conan server:

```
$ conan new mypackage/1.0@myuser/stable -t -ciglg -ciglc -ciu https://api.bintray.
↪com/conan/myuser/myrepo
```

conan upload

```
$ conan upload [-h] [-p PACKAGE] [-r REMOTE] [--all] [--skip-upload]
               [--force] [--check] [-c] [--retry RETRY]
               [--retry-wait RETRY_WAIT] [-no [{all,recipe}]] [-j JSON]
               pattern_or_reference
```

Uploads a recipe and binary packages to a remote. If no remote is specified, the first configured remote (by default conan-center, use 'conan remote list' to list the remotes) will be used.

```
positional arguments:
  pattern_or_reference  Pattern or package recipe reference, e.g., 'boost/*',
                        'MyPackage/1.2@user/channel'

optional arguments:
  -h, --help            show this help message and exit
  -p PACKAGE, --package PACKAGE
                        package ID to upload
  -r REMOTE, --remote REMOTE
                        upload to this specific remote
  --all                 Upload both package recipe and packages
  --skip-upload         Do not upload anything, just run the checks and the
                        compression
  --force               Do not check conan recipe date, override remote with
                        local
  --check               Perform an integrity check, using the manifests,
```

(continues on next page)

(continued from previous page)

```
                        before upload
  -c, --confirm         Upload all matching recipes without confirmation
  --retry RETRY         In case of fail retries to upload again the specified
                        times. Defaulted to 2
  --retry-wait RETRY_WAIT
                        Waits specified seconds before retry again
  -no [{all,recipe}], --no-overwrite [{all,recipe}]
                        Uploads package only if recipe is the same as the
                        remote one
  -j JSON, --json JSON  json file path where the install information will be
                        written to
```

Examples:

Uploads a package recipe (conanfile.py and the exported files):

```
$ conan upload OpenCV/1.4.0@lasote/stable
```

Uploads a package recipe and all the generated binary packages to a specified remote:

```
$ conan upload OpenCV/1.4.0@lasote/stable --all -r my_remote
```

Uploads all recipes and binary packages from our local cache to `my_remote` without confirmation:

```
$ conan upload "*" --all -r my_remote -c
```

Upload all local packages and recipes beginning with "Op" retrying 3 times and waiting 10 seconds between upload attempts:

```
$ conan upload "Op*" --all -r my_remote -c --retry 3 --retry_wait 10
```

Upload packages without overwriting the recipe and packages if the recipe has changed:

```
$ conan upload OpenCV/1.4.0@lasote/stable --all --no-overwrite  # defaults to --no-
↪overwrite all
```

Upload packages without overwriting the recipe if the packages has changed:

```
$ conan upload OpenCV/1.4.0@lasote/stable --all --no-overwrite recipe
```

conan test

```
$ conan test [-h] [-tbf TEST_BUILD_FOLDER] [-b [BUILD]] [-e ENV]
             [-o OPTIONS] [-pr PROFILE] [-r REMOTE] [-s SETTINGS] [-u]
             path reference
```

Test a package consuming it from a conanfile.py with a test() method. This command installs the conanfile dependencies (including the tested package), calls a 'conan build' to build test apps and finally executes the test() method. The testing recipe does not require name or version, neither definition of package() or package_info() methods. The package to be tested must exist in the local cache or in any configured remote.

```
positional arguments:
  path                  Path to the "testing" folder containing a conanfile.py
                        or to a recipe file with test() methode.g. conan
```

(continues on next page)

(continued from previous page)

```
                        test_package/conanfile.py pkg/version@user/channel
  reference             pkg/version@user/channel of the package to be tested

optional arguments:
  -h, --help            show this help message and exit
  -tbf TEST_BUILD_FOLDER, --test-build-folder TEST_BUILD_FOLDER
                        Working directory of the build process.
  -b [BUILD], --build [BUILD]
                        Optional, use it to choose if you want to build from
                        sources: --build Build all from sources, do not use
                        binary packages. --build=never Never build, use binary
                        packages or fail if a binary package is not found.
                        --build=missing Build from code if a binary package is
                        not found. --build=outdated Build from code if the
                        binary is not built with the current recipe or when
                        missing binary package. --build=[pattern] Build always
                        these packages from source, but never build the
                        others. Allows multiple --build parameters. 'pattern'
                        is a fnmatch file pattern of a package name. Default
                        behavior: If you don't specify anything, it will be
                        similar to '--build=never', but package recipes can
                        override it with their 'build_policy' attribute in the
                        conanfile.py.
  -e ENV, --env ENV     Environment variables that will be set during the
                        package build, -e CXX=/usr/bin/clang++
  -o OPTIONS, --options OPTIONS
                        Define options values, e.g., -o Pkg:with_qt=true
  -pr PROFILE, --profile PROFILE
                        Apply the specified profile to the install command
  -r REMOTE, --remote REMOTE
                        Look in the specified remote server
  -s SETTINGS, --settings SETTINGS
                        Settings to build the package, overwriting the
                        defaults. e.g., -s compiler=gcc
  -u, --update          Check updates exist from upstream remotes
```

This command is util for testing existing packages, that have been previously built (with **conan create**, for example). **conan create** will automatically run this test if a *test_package* folder is found besides the *conanfile.py*, or if the **--test-folder** argument is provided to **conan create**.

Example:

```
$ conan new Hello/0.1 -s -t
$ mv test_package test_package2
$ conan create . user/testing
# doesn't automatically run test, it has been renamed
# now run test
$ conan test test_package2 Hello/0.1@user/testing
```

The test package folder, could be elsewhere, or could be even applied to different versions of the package.

13.1.3 Package development commands

Commands related to the local (user space) development of a Conan package:

conan source

```
$ conan source [-h] [-sf SOURCE_FOLDER] [-if INSTALL_FOLDER] path
```

Calls your local conanfile.py 'source()' method. e.g., Downloads and unzip the package sources.

```
positional arguments:
  path                  Path to a folder containing a conanfile.py or to a
                        recipe file e.g., my_folder/conanfile.py

optional arguments:
  -h, --help            show this help message and exit
  -sf SOURCE_FOLDER, --source-folder SOURCE_FOLDER
                        Destination directory. Defaulted to current directory
  -if INSTALL_FOLDER, --install-folder INSTALL_FOLDER
                        Directory containing the conaninfo.txt and
                        conanbuildinfo.txt files (from previous 'conan
                        install'). Defaulted to --build-folder Optional,
                        source method will run without the information
                        retrieved from the conaninfo.txt and
                        conanbuildinfo.txt, only required when using
                        conditional source() based on settings, options,
                        env_info and user_info
```

The `source()` method might use (optional) *settings*, *options* and *environment variables* from the specified profile and dependencies information from the declared `deps_XXX_info` objects in the conanfile requirements.

All that information is saved automatically in the *conaninfo.txt* and *conanbuildinfo.txt* files respectively, when you run the **`conan install`** command. Those files have to be located in the specified **`--install-folder`**.

Examples:

- Call a local recipe's source method: In user space, the command will execute a local *conanfile.py* `source()` method, in the *src* folder in the current directory.

  ```
  $ conan new lib/1.0@conan/stable
  $ conan source . --source-folder mysrc
  ```

- In case you need the settings/options or any info from the requirements, perform first an install:

  ```
  $ conan install . --install-folder mybuild
  $ conan source . --source-folder mysrc --install-folder mybuild
  ```

conan build

```
$ conan build [-h] [-b] [-bf BUILD_FOLDER] [-c] [-i] [-if INSTALL_FOLDER]
              [-pf PACKAGE_FOLDER] [-sf SOURCE_FOLDER]
              path
```

Calls your local conanfile.py 'build()' method. The recipe will be built in the local directory specified by –build-folder, reading the sources from –source-folder. If you are using a build helper, like CMake(), the –package- folder will be configured as destination folder for the install step.

```
positional arguments:
  path                  Path to a folder containing a conanfile.py or to a
                        recipe file e.g., my_folder/conanfile.py
```

(continues on next page)

(continued from previous page)

```
optional arguments:
  -h, --help            show this help message and exit
  -b, --build           Execute the build step (variable should_build=True).
                        When specified, configure/install won't run unless
                        --configure/--install specified
  -bf BUILD_FOLDER, --build-folder BUILD_FOLDER
                        Directory for the build process. Defaulted to the
                        current directory. A relative path to current
                        directory can also be specified
  -c, --configure       Execute the configuration step (variable
                        should_configure=True). When specified, build/install
                        won't run unless --build/--install specified
  -i, --install         Execute the install step (variable
                        should_install=True). When specified, configure/build
                        won't run unless --configure/--build specified
  -if INSTALL_FOLDER, --install-folder INSTALL_FOLDER
                        Directory containing the conaninfo.txt and
                        conanbuildinfo.txt files (from previous 'conan
                        install'). Defaulted to --build-folder
  -pf PACKAGE_FOLDER, --package-folder PACKAGE_FOLDER
                        Directory to install the package (when the build
                        system or build() method does it). Defaulted to the
                        '{build_folder}/package' folder. A relative path can
                        be specified, relative to the current folder. Also an
                        absolute path is allowed.
  -sf SOURCE_FOLDER, --source-folder SOURCE_FOLDER
                        Directory containing the sources. Defaulted to the
                        conanfile's directory. A relative path to current
                        directory can also be specified
```

The `build()` method might use *settings*, *options* and *environment variables* from the specified profile and dependencies information from the declared `deps_XXX_info` objects in the conanfile requirements. All that information is saved automatically in the *conaninfo.txt* and *conanbuildinfo.txt* files respectively, when you run the **`conan install`** command. Those files have to be located in the specified **`--build-folder`** or in the **`--install-folder`** if specified.

The **`--configure`**, **`--build`**, **`--install`** arguments control which parts of the `build()` are actually executed. They have related conanfile boolean variables `should_configure`, `should_build`, `should_install`, which are `True` by default, but that will change if some of these arguments are used in the command line. The `CMake` and `Meson` and `AutotoolsBuildEnvironment` helpers already use these variables.

Example: Building a conan package (for architecture x86) in a local directory.

Listing 1: conanfile.py

```
from conans import ConanFile, CMake, tools

class LibConan(ConanFile):
    ...

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

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

(continues on next page)

(continued from previous page)

```
        cmake.build()
```

First we will call **conan source** to get our source code in the *src* directory, then **conan install** to install the requirements and generate the info files, and finally **conan build** to build the package:

```
$ conan source . --source-folder src
$ conan install . --install-folder build_x86 -s arch=x86
$ conan build . --build-folder build_x86 --source-folder src
```

Or if we want to create the `conaninfo.txt` and `conanbuildinfo.txt` files in a different folder:

```
$ conan source . --source-folder src
$ conan install . --install-folder install_x86 -s arch=x86
$ conan build . --build-folder build_x86 --install-folder install_x86 --source-
↪folder src
```

However, we recommend the `conaninfo.txt` and `conanbuildinfo.txt` to be generated in the same –build-folder, otherwise, you will need to specify a different folder in your build system to include the files generators file. e.j `conanbuildinfo.cmake`

Example: Control the build stages

Given a conanfile with this `build()` method:

```
def build(self):
    cmake = CMake(self)
    cmake.configure()
    cmake.build()
    cmake.install()
```

If nothing is specified, all three methods will be called. But using command line arguments, this can be changed:

```
$ conan build . -c # only run cmake.configure(). Other methods will do nothing
$ conan build . -b # only run cmake.build(). Other methods will do nothing
$ conan build . -i # only run cmake.install(). Other methods will do nothing
# They can be combined
$ conan build . -c -b # run cmake.configure() + cmake.build(), but not cmake.install()
```

Autotools and Meson helpers already implement the same functionality. For other build systems, you can use the following variables in the `build()` method:

```
def build(self):
    if self.should_configure:
        # Run my configure stage
    if self.should_build:
        # Run my build stage
    if self.should_install: # If my build has install, otherwise use package()
        # Run my install stage
```

Note these `should_configure`, `should_build`, `should_install` variables will always be `True` while building in the local cache. They can only be modified for the local flow with **conan build**.

conan package

```
$ conan package [-h] [-bf BUILD_FOLDER] [-if INSTALL_FOLDER]
                [-pf PACKAGE_FOLDER] [-sf SOURCE_FOLDER]
                path
```

Calls your local conanfile.py 'package()' method. This command works in the user space and it will copy artifacts from the –build-folder and –source- folder folder to the –package-folder one. It won't create a new package in the local cache, if you want to do it, use 'conan create' or 'conan export- pkg' after a 'conan build' command.

```
positional arguments:
  path                  Path to a folder containing a conanfile.py or to a
                        recipe file e.g., my_folder/conanfile.py

optional arguments:
  -h, --help            show this help message and exit
  -bf BUILD_FOLDER, --build-folder BUILD_FOLDER
                        Directory for the build process. Defaulted to the
                        current directory. A relative path to current
                        directory can also be specified
  -if INSTALL_FOLDER, --install-folder INSTALL_FOLDER
                        Directory containing the conaninfo.txt and
                        conanbuildinfo.txt files (from previous 'conan
                        install'). Defaulted to --build-folder
  -pf PACKAGE_FOLDER, --package-folder PACKAGE_FOLDER
                        folder to install the package. Defaulted to the
                        '{build_folder}/package' folder. A relative path can
                        be specified (relative to the current directory). Also
                        an absolute path is allowed.
  -sf SOURCE_FOLDER, --source-folder SOURCE_FOLDER
                        Directory containing the sources. Defaulted to the
                        conanfile's directory. A relative path to current
                        directory can also be specified
```

The `package()` method might use *settings*, *options* and *environment variables* from the specified profile and dependencies information from the declared `deps_XXX_info` objects in the conanfile requirements.

All that information is saved automatically in the *conaninfo.txt* and *conanbuildinfo.txt* files respectively, when you run **`conan install`**. Those files have to be located in the specified **`--build-folder`**.

```
$ conan install . --build-folder=build
```

Examples

This example shows how `package()` works in a package which can be edited and built in user folders instead of the local cache.

```
$ conan new Hello/0.1 -s
$ conan install . --install-folder=build_x86 -s arch=x86
$ conan build . --build-folder=build_x86
$ conan package . --build-folder=build_x86 --package-folder=package_x86
$ ls package/x86
> conaninfo.txt  conanmanifest.txt  include/  lib/
```

Note: The packages created locally are just for the user, but cannot be directly consumed by other packages, nor they can be uploaded to a remote repository. In order to make these packages available to the system, they have to be put in the conan local cache, which can be done with the **`conan export-pkg`** command instead of using **`conan package`** command:

```
$ conan new Hello/0.1 -s
$ conan install . --install-folder=build_x86 -s arch=x86
$ conan build . --build-folder=build_x86
$ conan export-pkg . Hello/0.1@user/stable --build-folder=build_x86 -s arch=x86
```

13.1.4 Misc commands

Other useful commands:

conan profile

```
$ conan profile [-h] {list,show,new,update,get,remove} ...
```

Lists profiles in the '.conan/profiles' folder, or shows profile details. The 'list' subcommand will always use the default user 'conan/profiles' folder. But the 'show' subcommand is able to resolve absolute and relative paths, as well as to map names to '.conan/profiles' folder, in the same way as the '– profile' install argument.

```
positional arguments:
  {list,show,new,update,get,remove}
    list                List current profiles
    show                Show the values defined for a profile
    new                 Creates a new empty profile
    update              Update a profile with desired value
    get                 Get a profile key
    remove              Remove a profile key

optional arguments:
  -h, --help            show this help message and exit
```

Examples

- List the profiles:

```
$ conan profile list
> myprofile1
> myprofile2
```

- Print profile contents:

```
$ conan profile show myprofile1
Profile myprofile1
[settings]
...
```

- Print profile contents (in the standard directory `.conan/profiles`):

```
$ conan profile show myprofile1
Profile myprofile1
[settings]
...
```

- Print profile contents (in a custom directory):

```
$ conan profile show /path/to/myprofile1
Profile myprofile1
[settings]
...
```

- Update a setting from a profile located in a custom directory:

```
$ conan profile update settings.build_type=Debug /path/to/my/profile
```

- Add a new option to the default profile:

```
$ conan profile update options.zlib:shared=True default
```

- Create a new empty profile:

```
$ conan profile new /path/to/new/profile
```

- Create a new profile detecting the settings:

```
$ conan profile new /path/to/new/profile --detect
```

conan remote

```
$ conan remote [-h]
               {list,add,remove,update,rename,list_ref,add_ref,remove_ref,update_ref}
               ...
```

Manages the remote list and the package recipes associated to a remote.

```
positional arguments:
  {list,add,remove,update,rename,list_ref,add_ref,remove_ref,update_ref}
                        sub-command help
    list                List current remotes
    add                 Add a remote
    remove              Remove a remote
    update              Update the remote url
    rename              Update the remote name
    list_ref            List the package recipes and its associated remotes
    add_ref             Associate a recipe's reference to a remote
    remove_ref          Dissociate a recipe's reference and its remote
    update_ref          Update the remote associated with a package recipe

optional arguments:
  -h, --help            show this help message and exit
```

Examples

- List remotes:

```
$ conan remote list
conan-center: https://conan.bintray.com [Verify SSL: True]
local: http://localhost:9300 [Verify SSL: True]
```

- Add a new remote:

```
$ conan remote add remote_name remote_url [verify_ssl]
```

Verify SSL option can be True or False (default True). Conan client will verify the SSL certificates.

- Insert a new remote:

  Insert as the first one (position/index 0), so it is the first one to be checked:

```
$ conan remote add remote_name remote_url [verify_ssl] --insert
```

  Insert as the second one (position/index 1), so it is the second one to be checked:

```
$ conan remote add remote_name remote_url [verify_ssl] --insert=1
```

- Add or insert a remote:

Adding the `--force` argument to `conan remote add` will always work, and won't raise an error. If an existing remote exists with that remote name or URL, it will be updated with the new information. The `--insert` works the same. If not specified, the remote will be appended the last one. If specified, the command will insert the remote in the specified position

```
$ conan remote add remote_name remote_url [verify_ssl] --force --insert=1
```

- Remove a remote:

```
$ conan remote remove remote_name
```

- Update a remote:

```
$ conan remote update remote_name new_url [verify_ssl]
```

- Rename a remote:

```
$ conan remote rename remote_name new_remote_name
```

- Change an existing remote to the first position:

```
$ conan remote update remote_name same_url --insert 0
```

- List the package recipes and its associated remotes:

```
$ conan remote list_ref
bzip2/1.0.6@lasote/stable: conan.io
Boost/1.60.0@lasote/stable: conan.io
zlib/1.2.8@lasote/stable: conan.io
```

- Associate a recipe's reference to a remote:

```
$ conan remote add_ref OpenSSL/1.0.2i@conan/stable conan-center
```

- Update the remote associated with a package recipe:

```
$ conan remote update_ref OpenSSL/1.0.2i@conan/stable local-remote
```

Note: Check the section *How to manage SSL (TLS) certificates* section to know more about server certificates verification and client certifications management .

conan user

```
$ conan user [-h] [-c] [-p [PASSWORD]] [-r REMOTE] [name]
```

Authenticates against a remote with user/pass, caching the auth token. Useful to avoid the user and password being requested later. e.g. while you're uploading a package. You can have one user for each remote. Changing the user, or introducing the password is only necessary to perform changes in remote packages.

```
positional arguments:
  name                  Username you want to use. If no name is provided it
                        will show the current user

optional arguments:
  -h, --help            show this help message and exit
  -c, --clean           Remove user and tokens for all remotes
  -p [PASSWORD], --password [PASSWORD]
                        User password. Use double quotes if password with
                        spacing, and escape quotes if existing. If empty, the
                        password is requested interactively (not exposed)
  -r REMOTE, --remote REMOTE
                        Use the specified remote server
```

Examples:

- List my user for each remote:

```
$ conan user
```

- Change **bar** remote user to **foo**:

```
$ conan user foo -r bar
```

- Change **bar** remote user to **foo**, authenticating against the remote and storing the user and authentication token locally, so a later upload won't require entering credentials:

```
$ conan user foo -r bar -p mypassword
```

- Clean all local users and tokens:

```
$ conan user --clean
```

- Change **bar** remote user to **foo**, **asking user password** to authenticate against the remote and storing the user and authentication token locally, so a later upload won't require entering credentials:

```
$ conan user foo -r bar -p
Please enter a password for "foo" account:
Change 'bar' user from None (anonymous) to foo
```

Note: The password is not stored in the client computer at any moment. Conan uses JWT, so it gets a token (expirable by the server) checking the password against the remote credentials. If the password is correct, an authentication token will be obtained, and that token is the information cached locally. For any subsequent interaction with the remotes, the conan client will only use that JWT token.

conan imports

```
$ conan imports [-h] [-if INSTALL_FOLDER] [-imf IMPORT_FOLDER] [-u] path
```

Calls your local conanfile.py or conanfile.txt 'imports' method. It requires to have been previously installed and have a conanbuildinfo.txt generated file in the –install-folder (defaulted to current directory).

```
positional arguments:
  path                  Path to a folder containing a conanfile.py or to a
                        recipe file e.g., my_folder/conanfile.py With --undo
                        option, this parameter is the folder containing the
                        conan_imports_manifest.txt file generated in a
                        previousexecution. e.j: conan imports ./imported_files
                        --undo

optional arguments:
  -h, --help            show this help message and exit
  -if INSTALL_FOLDER, --install-folder INSTALL_FOLDER
                        Directory containing the conaninfo.txt and
                        conanbuildinfo.txt files (from previous 'conan
                        install'). Defaulted to --build-folder
  -imf IMPORT_FOLDER, --import-folder IMPORT_FOLDER
                        Directory to copy the artifacts to. By default it will
                        be the current directory
  -u, --undo            Undo imports. Remove imported files
```

The `imports()` method might use *settings*, *options* and *environment variables* from the specified profile and dependencies information from the declared `deps_XXX_info` objects in the conanfile requirements.

All that information is saved automatically in the *conaninfo.txt* and *conanbuildinfo.txt* files respectively, when you run **`conan install`**. Those files have to be located in the specified **`--install-folder`**.

Examples

- Import files from a current conanfile in current directory:

```
$ conan install . --no-imports # Creates the conanbuildinfo.txt
$ conan imports .
```

- Remove the copied files (undo the import):

```
$ conan imports . --undo
```

conan copy

```
$ conan copy [-h] [-p PACKAGE] [--all] [--force] reference user_channel
```

Copies conan recipes and packages to another user/channel. Useful to promote packages (e.g. from "beta" to "stable") or transfer them from one user to another.

```
positional arguments:
  reference             package reference. e.g., MyPackage/1.2@user/channel
  user_channel          Destination user/channel. e.g., lasote/testing

optional arguments:
  -h, --help            show this help message and exit
```

(continues on next page)

(continued from previous page)

```
  -p PACKAGE, --package PACKAGE
                        copy specified package ID
  --all                 Copy all packages from the specified package recipe
  --force               Override destination packages and the package recipe
```

Examples

- Promote a package to **stable** from **beta**:

```
$ conan copy OpenSSL/1.0.2i@lasote/beta lasote/stable
```

- Change a package's username:

```
$ conan copy OpenSSL/1.0.2i@lasote/beta foo/beta
```

conan download

```
$ conan download [-h] [-p PACKAGE] [-r REMOTE] [-re] reference
```

Downloads recipe and binaries to the local cache, without using settings. It works specifying the recipe reference and package ID to be installed. Not transitive, requirements of the specified reference will NOT be retrieved. Useful together with 'conan copy' to automate the promotion of packages to a different user/channel. Only if a reference is specified, it will download all packages from the specified remote. Otherwise, it will search sequentially in the configured remotes.

```
positional arguments:
  reference             pkg/version@user/channel

optional arguments:
  -h, --help            show this help message and exit
  -p PACKAGE, --package PACKAGE
                        Force install specified package ID (ignore
                        settings/options)
  -r REMOTE, --remote REMOTE
                        look in the specified remote server
  -re, --recipe         Downloads only the recipe
```

Examples

- Download all **OpenSSL/1.0.2i@conan/stable** binary packages from the remote **foo**:

```
$ conan download OpenSSL/1.0.2i@conan/stable -r foo
```

- Download a single binary package of **OpenSSL/1.0.2i@conan/stable** from the remote **foo**:

```
$ conan download OpenSSL/1.0.2i@conan/stable -r foo -p␣
↪8018a4df6e7d2b4630a814fa40c81b85b9182d2
```

- Download only the recipe of package **OpenSSL/1.0.2i@conan/stable** from the remote **foo**:

```
$ conan download OpenSSL/1.0.2i@conan/stable -r foo -re
```

conan remove

```
$ conan remove [-h] [-b [BUILDS [BUILDS ...]]] [-f] [-o]
               [-p [PACKAGES [PACKAGES ...]]] [-q QUERY] [-r REMOTE] [-s]
               [-l]
               [pattern_or_reference]
```

Removes packages or binaries matching pattern from local cache or remote. It can also be used to remove temporary source or build folders in the local conan cache. If no remote is specified, the removal will be done by default in the local conan cache.

```
positional arguments:
  pattern_or_reference  Pattern or package recipe reference, e.g., 'boost/*',
                        'MyPackage/1.2@user/channel'

optional arguments:
  -h, --help            show this help message and exit
  -b [BUILDS [BUILDS ...]], --builds [BUILDS [BUILDS ...]]
                        By default, remove all the build folders or select
                        one, specifying the package ID
  -f, --force           Remove without requesting a confirmation
  -o, --outdated        Remove only outdated from recipe packages
  -p [PACKAGES [PACKAGES ...]], --packages [PACKAGES [PACKAGES ...]]
                        Select package to remove specifying the package ID
  -q QUERY, --query QUERY
                        Packages query: 'os=Windows AND (arch=x86 OR
                        compiler=gcc)'. The 'pattern_or_reference' parameter
                        has to be a reference: MyPackage/1.2@user/channel
  -r REMOTE, --remote REMOTE
                        Will remove from the specified remote
  -s, --src             Remove source folders
  -l, --locks           Remove locks
```

The `-q` parameter can't be used along with `-p` nor `-b` parameters.

Examples:

- Remove from the local cache the binary packages (the package recipes will not be removed) from all the recipes matching `OpenSSL/*` pattern:

```
$ conan remove OpenSSL/* --packages
```

- Remove the temporary build folders from all the recipes matching `OpenSSL/*` pattern without requesting confirmation:

```
$ conan remove OpenSSL/* --builds --force
```

- Remove the recipe and the binary packages from a specific remote:

```
$ conan remove OpenSSL/1.0.2@lasote/stable -r myremote
```

- Remove only Windows OpenSSL packages from local cache:

```
$ conan remove OpenSSL/1.0.2@lasote/stable -q "os=Windows"
```

conan alias

```
$ conan alias [-h] reference target
```

Creates and exports an 'alias package recipe'. An "alias" package is a symbolic name (reference) for another package (target). When some package depends on an alias, the target one will be retrieved and used instead, so the alias reference, the symbolic name, does not appear in the final dependency graph.

```
positional arguments:
  reference   Alias reference. e.j: mylib/1.X@user/channel
  target      Target reference. e.j: mylib/1.12@user/channel

optional arguments:
  -h, --help  show this help message and exit
```

The command:

```
$ conan alias Hello/0.X@user/testing Hello/0.1@user/testing
```

Creates and exports a package recipe for `Hello/0.X@user/testing` with the following content:

```
from conans import ConanFile

class AliasConanfile(ConanFile):
    alias = "Hello/0.1@user/testing"
```

Such package recipe acts as a "proxy" for the aliased reference. Users depending on `Hello/0.X@user/testing` will actually use version `Hello/0.1@user/testing`. The alias package reference will not appear in the dependency graph at all. It is useful to define symbolic names, or behaviors like "always depend on the latest minor", but defined upstream instead of being defined downstream with `version-ranges`.

The "alias" package should be uploaded to servers in the same way as regular package recipes, in order to enable usage from servers.

conan help

```
$ conan help [-h] [command]
```

Show help of a specific commmand.

```
positional arguments:
  command     command

optional arguments:
  -h, --help  show this help message and exit
```

This command is equivalent to the `--help` and `-h` arguments

Example:

```
$ conan help get
> usage: conan get [-h] [-p PACKAGE] [-r REMOTE] [-raw] reference [path]
> Gets a file or list a directory of a given reference or package.

# same as
$ conan get -h
```

13.1.5 Output

JSON documents generated by the commands:

Install and Create output [EXPERIMENTAL]

The **conan install** and **conan create** provide a `--json` parameter to generate a file containing the information of the installation process.

The output JSON contains a two first level keys:

- **error**: `True` if the install completed without error, `False` otherwise.
- **installed**: A list of installed packages. Each element contains:
  - **recipe**: Document representing the downloaded recipe.
    - **remote**: remote URL if the recipe has been downloaded. `null` otherwise.
    - **cache**: `true`/`false`. Retrieved from cache (not downloaded).
    - **downloaded**: `true`/`false`. Downloaded from a remote (not in cache).
    - **time**: `ISO 8601` string with the time the recipe was downloaded/retrieved.
    - **error**: `true`/`false`.
    - **id**: Reference. e.j: "OpenSSL/1.0.2n@conan/stable"
    - **dependency**: `true`/`false`. Is the package being installed/created or a dependency. Same as *develop conanfile attribute*.
  - **packages**: List of elements, representing the binary packages downloaded for the recipe. Normally there will be only 1 element in this list, only in special cases with build requires, private dependencies and settings overrided this list could have more than one element.
    - **remote**: remote URL if the recipe has been downloaded. `null` otherwise.
    - **cache**: `true`/`false`. Retrieved from cache (not downloaded).
    - **downloaded**: `true`/`false`. Downloaded from a remote (not in cache).
    - **time**: ISO 8601 string with the time the recipe was downloaded/retrieved.
    - **error**: `true`/`false`.
    - **id**: Package ID. e.j: "8018a4df6e7d2b4630a814fa40c81b85b9182d2b"

Example:

```
$ conan install OpenSSL/1.0.2n@conan/stable --json install.json
```

Listing 2: install.json

```
{
   "installed":[
      {
         "packages":[
            {
               "remote":null,
               "built":false,
               "cache":true,
               "downloaded":false,
```

(continues on next page)

(continued from previous page)


```
                "time":"2018-03-28T08:39:41.385285",
                "error":null,
                "id":"227fb0ea22f4797212e72ba94ea89c7b3fbc2a0c"
            }
        ],
        "recipe":{
            "remote":null,
            "cache":true,
            "downloaded":false,
            "time":"2018-03-28T08:39:41.365836",
            "error":null,
            "id":"OpenSSL/1.0.2n@conan/stable"
        }
    },
    {
        "packages":[
            {
                "remote":null,
                "built":false,
                "cache":true,
                "downloaded":false,
                "time":"2018-03-28T08:39:41.384952",
                "error":null,
                "id":"8018a4df6e7d2b4630a814fa40c81b85b9182d2b"
            }
        ],
        "recipe":{
            "remote":null,
            "cache":true,
            "downloaded":false,
            "time":"2018-03-28T08:39:41.379354",
            "error":null,
            "id":"zlib/1.2.11@conan/stable"
        }
    }
  ],
  "error":false
}
```


Upload output [EXPERIMENTAL]

The **conan upload** provides a `--json` parameter to generate a file containing the information of the upload process.

The output JSON contains a two first level keys:

- **error**: `True` if the upload completed without error, `False` otherwise.
- **uploaded**: A list of installed packages. Each element contains:
  - **recipe**: Document representing the uploaded recipe.
    - **id**: Reference. e.j: "OpenSSL/1.0.2n@conan/stable"
    - **remote_name**: Remote name where the recipe was uploaded.
    - **remote_url**: Remote URL where the recipe was uploaded.
    - **time**: `ISO 8601` string with the time the recipe was uploaded.

- **packages: List of elements, representing the binary packages uploaded for the recipe.**
  - **id**: Package ID. e.j: "8018a4df6e7d2b4630a814fa40c81b85b9182d2b"
  - **time**: ISO 8601 string with the time the recipe was uploaded.

Example:

```
$ conan upload h* -all -r conan-center --json upload.json
```

Listing 3: install.json


```
{
    "error":false,
    "uploaded":[
        {
            "recipe":{
                "id":"Hello/0.1@conan/testing",
                "remote_name":"conan-center",
                "remote_url":"https://conan.bintray.com",
                "time":"2018-04-30T11:18:19.204728"
            },
            "packages":[
                {
                    "id":"3f3387d49612e03a5306289405a2101383b861f0",
                    "time":"2018-04-30T11:18:21.534877"
                },
                {
                    "id":"6cc50b139b9c3d27b3e9042d5f5372d327b3a9f7",
                    "time":"2018-04-30T11:18:23.934152"
                },
                {
                    "id":"889d5d7812b4723bd3ef05693ffd190b1106ea43",
                    "time":"2018-04-30T11:18:28.195266"
                },
                {
                    "id":"e98aac15065fc710dffd1b4fbee382b087c3ad1d",
                    "time":"2018-04-30T11:18:30.495989"
                }
            ]
        },
        {
            "recipe":{
                "id":"Hello0/1.2.1@conan/testing",
                "remote_name":"conan-center",
                "remote_url":"https://conan.bintray.com",
                "time":"2018-04-30T11:18:32.688651"
            },
            "packages":[
                {
                    "id":"5ab84d6acfe1f23c4fae0ab88f26e3a396351ac9",
                    "time":"2018-04-30T11:18:34.991721"
                }
            ]
        },
        {
            "recipe":{
                "id":"HelloApp/0.1@conan/testing",
                "remote_name":"conan-center",
```


(continues on next page)

(continued from previous page)


```
                "remote_url":"https://conan.bintray.com",
                "time":"2018-04-30T11:18:36.901333"
            },
            "packages":[
                {
                    "id":"6cc50b139b9c3d27b3e9042d5f5372d327b3a9f7",
                    "time":"2018-04-30T11:18:39.243895"
                }
            ]
        },
        {
            "recipe":{
                "id":"HelloPythonConan/0.1@conan/testing",
                "remote_name":"conan-center",
                "remote_url":"https://conan.bintray.com",
                "time":"2018-04-30T11:18:41.181543"
            },
            "packages":[
                {
                    "id":"5ab84d6acfe1f23c4fae0ab88f26e3a396351ac9",
                    "time":"2018-04-30T11:18:43.749422"
                }
            ]
        },
        {
            "recipe":{
                "id":"HelloPythonReuseConan/0.1@conan/testing",
                "remote_name":"conan-center",
                "remote_url":"https://conan.bintray.com",
                "time":"2018-04-30T11:18:45.614096"
            },
            "packages":[
                {
                    "id":"6a051b2648c89dbd1f8ada0031105b287deea9d2",
                    "time":"2018-04-30T11:18:47.942491"
                }
            ]
        },
        {
            "recipe":{
                "id":"hdf5/1.8.20@acri/testing",
                "remote_name":"conan-center",
                "remote_url":"https://conan.bintray.com",
                "time":"2018-04-30T11:18:48.291756"
            },
            "packages":[

            ]
        },
        {
            "recipe":{
                "id":"http_parser/2.8.0@conan/testing",
                "remote_name":"conan-center",
                "remote_url":"https://conan.bintray.com",
                "time":"2018-04-30T11:18:48.637576"
            },
            "packages":[
```


(continues on next page)

(continued from previous page)

```
                    {
                        "id":"6cc50b139b9c3d27b3e9042d5f5372d327b3a9f7",
                        "time":"2018-04-30T11:18:51.125189"
                    }
                ]
            }
        ]
    }
```

13.2 conanfile.txt

13.2.1 Sections

[requires]

List of requirements, specifing the full reference.

```
[requires]
Poco/1.9.0@pocoproject/stable
zlib/1.2.11@conan/stable
```

Also support *version ranges*:

```
[requires]
Poco/[>1.0,<1.8]@pocoproject/stable
zlib/1.2.11@conan/stable
```

[build_requires]

List of build requirements, specifing the full reference.

```
[build_requires]
7z_installer/1.0@conan/stable
```

Also support *version ranges*

In practice the `[build_requires]` will be always installed, the same as `[requires]`. Installing from a *conanfile.txt* means that something is going to be built, so the build requirements are indeed needed.

Still, it is useful and conceptually cleaner to have them in separate sections, so users of this *conanfile.txt* might quickly identify some dev-tools that they have already installed on their machine, differentiating them from the required libraries to link with.

[generators]

List of *generators*

```
[requires]
Poco/1.9.0@pocoproject/stable
zlib/1.2.11@conan/stable
```

(continues on next page)

(continued from previous page)

```
[generators]
xcode
cmake
qmake
```

[options]

List of *options*. Always specifying **package_name:option = Value**

```
[requires]
Poco/1.9.0@pocoproject/stable
zlib/1.2.11@conan/stable

[generators]
cmake

[options]
Poco:shared=True
OpenSSL:shared=True
```

[imports]

List of files to be imported to a local directory. Read more: *imports*.

```
[requires]
Poco/1.9.0@pocoproject/stable
zlib/1.2.11@conan/stable

[generators]
cmake

[options]
Poco:shared=True
OpenSSL:shared=True

[imports]
bin, *.dll -> ./bin # Copies all dll files from packages bin folder to my local "bin"␣
↪folder
lib, *.dylib* -> ./bin # Copies all dylib files from packages lib folder to my local
↪"bin" folder
```

The first item is the subfolder of the packages (could be the root "." one), the second is the pattern to match. Both relate to the local cache. The third (after the arrow) item, is the destination folder, living in user space, not in the local cache.

The `[imports]` section also support the same arguments as the equivalent `imports()` method in *conanfile.py*, separated with an `@`.

- **root_package** (Optional, Defaulted to *all packages in deps*): fnmatch pattern of the package name ("OpenCV", "Boost") from which files will be copied.
- **folder**: (Optional, Defaulted to `False`). If enabled, it will copy the files from the local cache to a subfolder named as the package containing the files. Useful to avoid conflicting imports of files with the same name (e.g. License).

- **ignore_case**: (Optional, Defaulted to `False`). If enabled will do a case-insensitive pattern matching.
- **excludes**: (Optional, Defaulted to `None`). Allows defining a list of patterns (even a single pattern) to be excluded from the copy, even if they match the main `pattern`.
- **keep_path** (Optional, Defaulted to `True`): Means if you want to keep the relative path when you copy the files from the **src** folder to the **dst** one. Useful to ignore (`keep_path=False`) path of *library.dll* files in the package it is imported from.

Example to collect license files from dependencies, into a *licenses* folder, excluding (just an example) html and jpeg files:

```
[imports]
., license* -> ./licenses @ folder=True, ignore_case=True, excludes=*.html *.jpeg
```

13.3 conanfile.py

Reference for *conanfile.py*: attributes, methods, etc.

Contents:

13.3.1 Attributes

name

This is a string, with a minimun of 2 and a maximum of 50 characters (though shorter names are recommended), that defines the package name. It will be the `<PkgName>/version@user/channel` of the package reference. It should match the following regex `^[a-zA-Z0-9_][a-zA-Z0-9_\+\.-]$`, so start with alphanumeric or underscore, then alphanumeric, underscore, +, ., - characters.

The name is only necessary for `export`-ing the recipe into the local cache (`export` and `create` commands), if they are not defined in the command line. It might take its value from an environment variable, or even any python code that defines it (e.g. a function that reads an environment variable, or a file from disk). However, the most common and suggested approach would be to define it in plain text as a constant, or provide it as command line arguments.

version

The version attribute will define the version part of the package reference: `PkgName/<version>@user/channel` It is a string, and can take any value, matching the same constraints as the `name` attribute. In case the version follows semantic versioning in the form `X.Y.Z-pre1+build2`, that value might be used for requiring this package through version ranges instead of exact versions.

The version is only strictly necessary for `export`-ing the recipe into the local cache (`export` and `create` commands), if they are not defined in the command line. It might take its value from an environment variable, or even any python code that defines it (e.g. a function that reads an environment variable, or a file from disk). Please note that this value might be used in the recipe in other places (as in `source()` method to retrieve code from elsewhere), making this value not constant means that it may evaluate differently in different contexts (e.g., on different machines or for different users) leading to unrepeatable or unpredictable results. The most common and suggested approach would be to define it in plain text as a constant, or provide it as command line arguments.

description

This is an optional, but strongly recommended text field, containing the description of the package, and any information that might be useful for the consumers. The first line might be used as a short description of the package.

```
class HelloConan(ConanFile):
    name = "Hello"
    version = "0.1"
    description = """This is a Hello World library.
                     A fully featured, portable, C++ library to say Hello World in the␣
↪stdout,
                     with incredible iostreams performance"""
```

homepage

Use this attribute to indicate the home web page of the library being packaged. This is useful to link the recipe to further explanations of the library itself like an overview of its features, documentation, FAQ as well as other related information.

```
class EigenConan(ConanFile):
    name = "eigen"
    version = "3.3.4"
    homepage = "http://eigen.tuxfamily.org"
```

url

It is possible, even typical, if you are packaging a thid party lib, that you just develop the packaging code. Such code is also subject to change, often via collaboration, so it should be stored in a VCS like git, and probably put on GitHub or a similar service. If you do indeed maintain such a repository, please indicate it in the `url` attribute, so that it can be easily found.

```
class HelloConan(ConanFile):
    name = "Hello"
    version = "0.1"
    url = "https://github.com/memsharded/hellopack.git"
```

The `url` is the url **of the package** repository, i.e. not necessarily the original source code. It is optional, but highly recommended, that it points to GitHub, Bitbucket or your preferred code collaboration platform. Of course, if you have the conanfile inside your library source, you can point to it, and afterwards use the `url` in your `source()` method.

This is a recommended, but not mandatory attribute.

license

This field is intended for the license of the **target** source code and binaries, i.e. the code that is being packaged, not the `conanfile.py` itself. This info is used to be displayed by the **`conan info`** command and possibly other search and report tools.

```
class HelloConan(ConanFile):
    name = "Hello"
    version = "0.1"
    license = "MIT"
```

This attribute can contain several, comma separated licenses. It is a text string, so it can contain any text, including hyperlinks to license files elsewhere.

This is a recommended, but not mandatory attribute.

author

Intended to add information about the author, in case it is different from the conan user. It is possible that the conan user is the name of an organization, project, company or group, and many users have permissions over that account. In this case, the author information can explicitely define who is the creator/maintainer of the package

```
class HelloConan(ConanFile):
    name = "Hello"
    version = "0.1"
    author = "John J. Smith (john.smith@company.com)"
```

This is an optional attribute

user, channel

The fields `user` and `channel` can be accessed from within a `conanfile.py`. Though their usage is usually not encouraged, it could be useful in different cases, e.g. to define requirements with the same user and channel than the current package, which could be achieved with something like:

```
from conans import ConanFile

class HelloConan(ConanFile):
    name = "Hello"
    version = "0.1"

    def requirements(self):
        self.requires("Say/0.1@%s/%s" % (self.user, self.channel))
```

Only package recipes that are in the conan local cache (i.e. "exported") have an user/channel assigned. For package recipes working in user space, there is no current user/channel. The properties `self.user` and `self.channel` will then look for environment variables `CONAN_USERNAME` and `CONAN_CHANNEL` respectively. If they are not defined, an error will be raised.

settings

There are several things that can potentially affect a package being created, i.e. the final package will be different (a different binary, for example), if some input is different.

Development project-wide variables, like the compiler, its version, or the OS itself. These variables have to be defined, and they cannot have a default value listed in the conanfile, as it would not make sense.

It is obvious that changing the OS produces a different binary in most cases. Changing the compiler or compiler version changes the binary too, which might have a compatible ABI or not, but the package will be different in any case.

For these reasons, the most common convention among Conan recipes is to distinguish binaries by the following four settings, which is reflected in the *conanfile.py* template used in the *conan new* command:

```
settings = "os", "compiler", "build_type", "arch"
```

When Conan generates a compiled binary for a package with a given combination of the settings above, it generates a unique ID for that binary by hashing the current values of these settings.

But what happens for example to **header only libraries**? The final package for such libraries is not binary and, in most cases it will be identical, unless it is automatically generating code. We can indicate that in the conanfile:

```
from conans import ConanFile

 class HelloConan(ConanFile):
     name = "Hello"
     version = "0.1"
     # We can just omit the settings attribute too
     settings = None

     def build(self):
         #empty too, nothing to build in header only
```

You can restrict existing settings and accepted values as well, by redeclaring the settings attribute:

```
class HelloConan(ConanFile):
    settings = {"os": ["Windows"],
        "compiler": {"Visual Studio": {"version": [11, 12]}},
        "arch": None}
```

In this example we have just defined that this package only works in Windows, with VS 10 and 11. Any attempt to build it in other platforms with other settings will throw an error saying so. We have also defined that the runtime (the MD and MT flags of VS) is irrelevant for us (maybe we using a universal one?). Using None as a value means, *maintain the original values* in order to avoid re-typing them. Then, "arch": None is totally equivalent to "arch": ["x86", "x86_64", "arm"] Check the reference or your ~/.conan/settings.yml file.

As re-defining the whole settings attribute can be tedious, it is sometimes much simpler to remove or tune specific fields in the `configure()` method. For example, if our package is runtime independent in VS, we can just remove that setting field:

```
settings = "os", "compiler", "build_type", "arch"

def configure(self):
    self.settings.compiler["Visual Studio"].remove("runtime")
```

options, default_options

Conan packages recipes can generate different binary packages when different settings are used, but can also customize, per-package any other configuration that will produce a different binary.

A typical option would be being shared or static for a certain library. Note that this is optional, different packages can have this option, or not (like header-only packages), and different packages can have different values for this option, as opposed to settings, which typically have the same values for all packages being installed (though this can be controlled too, defining different settings for specific packages)

Options are defined in package recipes as dictionaries of name and allowed values:

```
class MyPkg(ConanFile):
    ...
    options = {"shared": [True, False]}
```

There is an special value `ANY` to allow any value for a given option. The range of values for such an option will not be checked, and any value (as string) will be accepted:

```
class MyPkg(ConanFile):
    ...
    options = {"shared": [True, False], "commit": "ANY"}
```

When a package is installed, it will need all its options be defined a value. Those values can be defined in command line, profiles, but they can also (and they will be typically) defined in conan package recipes:

```
class MyPkg(ConanFile):
    ...
    options = {"shared": [True, False], "fPIC": [True, False]}
    default_options = "shared=False", "fPIC=False"
```

The options will typically affect the `build()` of the package in some way, for example:

```
class MyPkg(ConanFile):
    ...
    options = {"shared": [True, False]}
    default_options = "shared=False"

    def build(self):
        shared = "-DBUILD_SHARED_LIBS=ON" if self.options.shared else ""
        cmake = CMake(self)
        self.run("cmake . %s %s" % (cmake.command_line, shared))
        self.run("cmake --build . %s" % cmake.build_config)
```

Note that you have to consider the option properly in your build scripts. In this case, we are using the CMake way. So if you had explicit **STATIC** linkage in the **CMakeLists.txt** file, you have to remove it. If you are using VS, you also need to change your code to correctly import/export symbols for the dll.

This is only an example. Actually, the `CMake` helper already automates this, so it is enough to do:

```
def build(self):
    cmake = CMake(self) # internally it will check self.options.shared
    self.run("cmake . %s" % cmake.command_line) # or cmake.configure()
    self.run("cmake --build . %s" % cmake.build_config) # or cmake.build()
```

You can also specify default option values of the required dependencies:

```
class OtherPkg(ConanFile):
    requires = "Pkg/0.1@user/channel"
    default_options = "Pkg:pkg_option=value"
```

You can also specify default option values of the conditional required dependencies:

```
class OtherPkg(ConanFile):
    default_options = "Pkg:pkg_option=value"

    def requirements(self):
        if self.settings.os != "Windows":
            self.requires("Pkg/0.1@user/channel")
```

This will always work, on Windows the *default_options* for the *Pkg/0.1@user/channel* will be ignored, they will only be used on every other os.

If you need to dynamically set some dependency options, you could do:

```
class OtherPkg(ConanFile):
    requires = "Pkg/0.1@user/channel"

    def configure(self):
        self.options["Pkg"].pkg_option = "value"
```

Option values can be given in command line, and they will have priority over the default values in the recipe:

```
$ conan install -o Pkg:shared=True -o OtherPkg:option=value
```

You can also defined them in consumer `conanfile.txt`, as described in *this section*

```
[requires]
Poco/1.9.0@pocoproject/stable

[options]
Poco:shared=True
OpenSSL:shared=True
```

And finally, you can define options in *profiles* too, with the same syntax:

```
# file "myprofile"
# use it as $ conan install -pr=myprofile
[settings]
setting=value

[options]
MyLib:shared=True
```

You can inspect available package options, reading the package recipe, which is conveniently done with:

```
$ conan get Pkg/0.1@user/channel
```

requires

Specify package dependencies as a list of other packages:

```
class MyLibConan(ConanFile):
    requires = "Hello/1.0@user/stable", "OtherLib/2.1@otheruser/testing"
```

You can specify further information about the package requirements:

```
class MyLibConan(ConanFile):
    requires = (("Hello/0.1@user/testing"),
                ("Say/0.2@dummy/stable", "override"),
                ("Bye/2.1@coder/beta", "private"))
```

Requirements can be complemented by 2 different parameters:

private: a dependency can be declared as private if it is going to be fully embedded and hidden from consumers of the package. Typical examples could be a header only library which is not exposed through the public interface of the package, or the linking of a static library inside a dynamic one, in which the functionality or the objects of the linked static library are not exposed through the public interface of the dynamic library.

override: packages can define overrides of their dependencies, if they require the definition of specific versions of the upstream required libraries, but not necessarily direct dependencies. For example, a package can depend on A(v1.0),

which in turn could conditionally depend on Zlib(v2), depending on whether the compression is enabled or not. Now, if you want to force the usage of Zlib(v3) you can:

```
class HelloConan(ConanFile):
    requires = ("A/1.0@user/stable", ("Zlib/3.0@other/beta", "override"))
```

This **will not introduce a new dependency**, it will just change Zlib v2 to v3 if A actually requires it. Otherwise Zlib will not be a dependency of your package.

version ranges

The syntax is using brackets:

```
class HelloConan(ConanFile):
    requires = "Pkg/[>1.0,<1.8]@user/stable"
```

Expressions are those defined and implemented by [python node-semver](https://pypi.org/project/node-semver/), but using a comma instead of spaces. Accepted expressions would be:

```
>1.1,<2.1     # In such range
2.8           # equivalent to =2.8
~=3.0         # compatible, according to semver
>1.1 || 0.8   # conditions can be OR'ed
```

Go to *Mastering/Version Ranges* if you want to learn more about version ranges.

build_requires

Build requirements are requirements that are only installed and used when the package is built from sources. If there is an existing pre-compiled binary, then the build requirements for this package will not be retrieved.

They can be specified as a comma separated tuple in the package recipe:

```
class MyPkg(ConanFile):
    build_requires = "ToolA/0.2@user/testing", "ToolB/0.2@user/testing"
```

Read more: *Build requiremens*

exports

If a package recipe `conanfile.py` requires other external files, like other python files that it is importing (python importing), or maybe some text file with data it is reading, those files must be exported with the `exports` field, so they are stored together, side by side with the `conanfile.py` recipe.

The `exports` field can be one single pattern, like `exports="*"`, or several inclusion patterns. For example, if we have some python code that we want the recipe to use in a `helpers.py` file, and have some text file, `info.txt`, we want to read and display during the recipe evaluation we would do something like:

```
exports = "helpers.py", "info.txt"
```

Exclude patterns are also possible, with the `!` prefix:

```
exports = "*.py", "!*tmp.py"
```

This is an optional attribute, only to be used if the package recipe requires these other files for evaluation of the recipe.

exports_sources

There are 2 ways of getting source code to build a package. Using the `source()` recipe method and using the `exports_sources` field. With `exports_sources` you specify which sources are required, and they will be exported together with the **conanfile.py**, copying them from your folder to the local conan cache. Using `exports_sources` the package recipe can be self-contained, containing the source code like in a snapshot, and then not requiring downloading or retrieving the source code from other origins (git, download) with the `source()` method when it is necessary to build from sources.

The `exports_sources` field can be one single pattern, like `exports_sources="*"`, or several inclusion patterns. For example, if we have the source code inside "include" and "src" folders, and there are other folders that are not necessary for the package recipe, we could do:

```
exports_sources = "include*", "src*"
```

Exclude patterns are also possible, with the `!` prefix:

```
exports_sources = "include*", "src*", "!src/build/*"
```

This is an optional attribute, used typically when `source()` is not specified. The main difference with `exports` is that `exports` files are always retrieved (even if pre-compiled packages exist), while `exports_sources` files are only retrieved when it is necessary to build a package from sources.

generators

Generators specify which is the output of the `install` command in your project folder. By default, a *conanbuildinfo.txt* file is generated, but you can specify different generators and even use more than one.

```
class MyLibConan(ConanFile):
    generators = "cmake", "gcc"
```

Check the full *generators list*.

build_policy

With the `build_policy` attribute the package creator can change the default conan's build behavior. The allowed `build_policy` values are:

- `missing`: If no binary package is found, conan will build it without the need of invoke conan install with **–build missing** option.
- `always`: The package will be built always, **retrieving each time the source code** executing the "source" method.

```
class PocoTimerConan(ConanFile):
    build_policy = "always" # "missing"
```

short_paths

If one of the packages you are creating hits the limit of 260 chars path length in Windows, add `short_paths=True` in your conanfile.py:

```
from conans import ConanFile

class ConanFileTest(ConanFile):
    ...
    short_paths = True
```

This will automatically "link" the `source` and `build` directories of the package to the drive root, something like *C:/.conan/tmpdir*. All the folder layout in the conan cache is maintained.

This attribute will not have any effect in other OS, it will be discarded.

From Windows 10 (ver. 10.0.14393), it is possible to opt-in disabling the path limits. Check this link for more info. Latest python installers might offer to enable this while installing python. With this limit removed, the `short_paths` functionality is totally unnecessary. Please note that this only works with Python 3.6 and newer.

no_copy_source

The attribute `no_copy_source` tells the recipe that the source code will not be copied from the `source` folder to the `build` folder. This is mostly an optimization for packages with large source codebases, to avoid extra copies. It is **mandatory** that the source code must not be modified at all by the configure or build scripts, as the source code will be shared among all builds.

To be able to use it, the package recipe can access the `self.source_folder` attribute, which will point to the `build` folder when `no_copy_source=False` or not defined, and will point to the `source` folder when `no_copy_source=True`

When this attribute is set to True, the `package()` method will be called twice, one copying from the `source` folder and the other copying from the `build` folder.

folders

In the package recipe methods, some attributes pointing to the relevant folders can be defined. Not all of them will be defined always, only in those relevant methods that might use them.

- `self.source_folder`: the folder in which the source code to be compiled lives. When a package is built in the conan local cache, by default it is the `build` folder, as the source code is copied from the `source` folder to the `build` folder, to ensure isolation and avoiding modifications of shared common source code among builds for different configurations. Only when `no_copy_source=True` this folder will actually point to the package `source` folder in the local cache.
- `self.build_folder`: the folder in which the build is being done
- `self.install_folder`: the folder in which the install has outputed the generator files, by default, and always in the local cache, is the same `self.build_folder`
- `self.package_folder`: the folder to copy the final artifacts for the binary package

When executing local conan commands (for a package not in the local cache, but in user folder), those fields would be pointing to the corresponding local user folder.

cpp_info

This attribute is only defined inside `package_info()` method, being None elsewhere, so please use it only inside this method.

The `self.cpp_info` object can be filled with the needed information for the consumers of the current package:

NAME	DESCRIPTION
self.cpp_info.includedirs	Ordered list with include paths, by default ['include']
self.cpp_info.libdirs	Ordered list with lib paths, by default ['lib']
self.cpp_info.resdirs	Ordered list of resource (data) paths, by default ['res']
self.cpp_info.bindirs	Ordered list with include paths, by default ['bin']
self.cpp_info.builddirs	Ordered list with build scripts paths, by default ['']
self.cpp_info.libs	Ordered list with the library names, by default empty []
self.cpp_info.defines	Preprocessor definitions, by default empty []
self.cpp_info.cflags	Ordered list with pure C flags, by default empty []
self.cpp_info.cppflags	Ordered list with C++ flags, by default empty []
self.cpp_info.sharedlinkflags	Ordered list with linker flags (shared libs), by default empty []
self.cpp_info.exelinkflags	Ordered list with linker flags (executables), by default empty []
self.cpp_info.rootpath	Filled with the root directory of the package, see deps_cpp_info

See also:

Read *package_info() method docs* for more info.

deps_cpp_info

Contains the `cpp_info` object of the requirements of the recipe. In addition of the above fields, there are also properties to obtain the absolute paths:

NAME	DESCRIPTION
self.cpp_info.include_paths	Same as `includedirs` but transformed to absolute paths
self.cpp_info.lib_paths	Same as `libdirs` but transformed to absolute paths
self.cpp_info.bin_paths	Same as `bindirs` but transformed to absolute paths
self.cpp_info.build_paths	Same as `builddirs` but transformed to absolute paths
self.cpp_info.res_paths	Same as `resdirs` but transformed to absolute paths

To get a list of all the dependency names from `deps_cpp_info`, you can call the *deps* member:

```
class PocoTimerConan(ConanFile):
    ...
    def build(self):
        # deps is a list of package names: ["Poco", "zlib", "OpenSSL"]
        deps = self.deps_cpp_info.deps
```

It can be used to get information about the dependencies, like used compilation flags or the root folder of the package:

```
class PocoTimerConan(ConanFile):
    ...
    requires = "zlib/1.2.11@conan/stable", "OpenSSL/1.0.2l@conan/stable"
    ...

    def build(self):
        # Get the directory where zlib package is installed
        self.deps_cpp_info["zlib"].rootpath

        # Get the absolute paths to zlib include directories (list)
        self.deps_cpp_info["zlib"].include_paths
```

(continues on next page)

(continued from previous page)

```
        # Get the sharedlinkflags property from OpenSSL package
        self.deps_cpp_info["OpenSSL"].sharedlinkflags
```

env_info

This attribute is only defined inside `package_info()` method, being None elsewhere, so please use it only inside this method.

The `self.env_info` object can be filled with the environment variables to be declared in the packages reusing the recipe.

See also:

Read *package_info() method docs* for more info.

deps_env_info

You can access to the declared environment variables of the requirements of the recipe.

Note: The environment variables declared in the requirements of a recipe are automatically applied and it can be accesed with the python `os.environ` dictionary. Nevertheless if you want to access to the variable declared by some specific requirement you can use the `self.deps_env_info` object.

```
import os

class RecipeConan(ConanFile):
    ...
    requires = "package1/1.0@conan/stable", "package2/1.2@conan/stable"
    ...

    def build(self):
        # Get the SOMEVAR environment variable declared in the "package1"
        self.deps_env_info["package1"].SOMEVAR

        # Access to the environment variables globally
        os.environ["SOMEVAR"]
```

info

Object used to control the unique ID for a package. Check the *package_id()* to see the details of the `self.info` object.

apply_env

When `True` (Default), the values from `self.deps_env_info` (corresponding to the declared `env_info` in the `requires` and `build_requires`) will be automatically applied to the `os.environ`.

Disable it setting `apply_env` to False if you want to control by yourself the environment variables applied to your recipes.

You can apply manually the environment variables from the requires and build_requires:

```
import os
from conans import tools

class RecipeConan(ConanFile):
    apply_env = False

    def build(self):
        with tools.environment_append(self.env):
            # The same if we specified apply_env = True
            pass
```

in_local_cache

A boolean attribute useful for conditional logic to apply in user folders local commands. It will return *True* if the conanfile resides in the local cache (we are installing the package) and *False* if we are running the conanfile in a user folder (local Conan commands).

```
import os

class RecipeConan(ConanFile):
    ...

    def build(self):
        if self.in_local_cache:
            # we are installing the package
        else:
            # we are building the package in a local directory
```

develop

A boolean attribute useful for conditional logic. It will be `True` if the package is created with **`conan create`**, or if the *conanfile.py* is in user space:

```
class RecipeConan(ConanFile):

    def build(self):
        if self.develop:
            self.output.info("Develop mode")
```

It can be used for conditional logic in other methods too, like `requirements()`, `package()`, etc.

This recipe will output "Develop mode" if:

```
$ conan create user/testing
# or
$ mkdir build && cd build && conan install ..
$ conan build ..
```

But it will not output that when it is a transitive requirement or installed with **`conan install`**.

keep_imports

Just before the `build()` method is executed, if the conanfile has an `imports()` method, it is executed into the build folder, to copy binaries from dependencies that might be necessary for the `build()` method to work. After the

method finishes, those copied (imported) files are removed, so they are not later unnecessarily repackaged.

This behavior can be avoided declaring the `keep_imports=True` attribute. This can be useful, for example to *repackage artifacts*

13.3.2 Methods

source()

Method used to retrieve the source code from any other external origin like github using `$ git clone` or just a regular download.

For example, "exporting" the source code files, together with the *conanfile.py* file, can be handy if the source code is not under version control. But if the source code is available in a repository, you can directly get it from there:

```
from conans import ConanFile

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

    def source(self):
        self.run("git clone https://github.com/memsharded/hello.git")
        # You can also change branch, commit or whatever
        # self.run("cd hello && git checkout 2fe5...")
```

This will work, as long as git is in your current path (so in Win you probably want to run things in msysgit, cmder, etc). You can also use another VCS or direct download/unzip. For that purpose, we have provided some helpers, but you can use your own code or origin as well. This is a snippet of the conanfile of the Poco library:

```
from conans import ConanFile
from conans.tools import download, unzip, check_md5, check_sha1, check_sha256
import os
import shutil

class PocoConan(ConanFile):
    name = "Poco"
    version = "1.6.0"

    def source(self):
        zip_name = "poco-1.6.0-release.zip"
        download("https://github.com/pocoproject/poco/archive/poco-1.6.0-release.zip",
↪ zip_name)
        # check_md5(zip_name, "51e11f2c02a36689d6ed655b6fff9ec9")
        # check_sha1(zip_name, "8d87812ce591ced8ce3a022beec1df1c8b2fac87")
        # check_sha256(zip_name,
↪"653f983c30974d292de58444626884bee84a2731989ff5a336b93a0fef168d79")
        unzip(zip_name)
        shutil.move("poco-poco-1.6.0-release", "poco")
        os.unlink(zip_name)
```

The download, unzip utilities can be imported from conan, but you can also use your own code here to retrieve source code from any origin. You can even create packages for pre-compiled libraries you already have, even if you don't have the source code. You can download the binaries, skip the `build()` method and define your `package()` and `package_info()` accordingly.

You can also use `check_md5()`, `check_sha1()` and `check_sha256()` from the *tools* module to verify that a package is downloaded correctly.

Note: It is very important to recall that the `source()` method will be executed just once, and the source code will be shared for all the package builds. So it is not a good idea to conditionally use settings or options to make changes or patches on the source code. Maybe the only setting that makes sense is the OS `self.settings.os`, if not doing cross-building, for example to retrieve different sources:

```
def source(self):
    if platform.system() == "Windows":
        # download some Win source zip
    else:
        # download sources from Nix systems in a tgz
```

If you need to patch the source code or build scripts differently for different variants of your packages, you can do it in the `build()` method, which uses a different folder and source code copy for each variant.

---

build()

This method is used to build the source code of the recipe using the desired commands. You can use your command line tools to invoke your build system or any of the build helpers provided with Conan.

```
def build(self):
    cmake = CMake(self)
    self.run("cmake . %s" % (cmake.command_line))
    self.run("cmake --build . %s" % cmake.build_config)
```

Build helpers

You can use these classes to prepare your build system's command invocation:

- **CMake**: Prepares the invocation of cmake command with your settings.
- **AutoToolsBuildEnvironment**: If you are using configure/Makefile to build your project you can use this helper. Read more: *Building with Autotools*.
- **MSBuild**: If you are using Visual Studio compiler directly to build your project you can use this helper *MSBuild()*. For lower level control, the **VisualStudioBuildEnvironment** can also be used: *VisualStudioBuildEnvironment*.

(Unit) Testing your library

We have seen how to run package tests with conan, but what if we want to run full unit tests on our library before packaging, so that they are run for every build configuration? Nothing special is required here. We can just launch the tests from the last command in our `build()` method:

```
def build(self):
    cmake = CMake(self)
    cmake.configure()
    cmake.build()
    # here you can run CTest, launch your binaries, etc
    cmake.test()
```

package()

The actual creation of the package, once that it is built, is done in the `package()` method. Using the `self.copy()` method, artifacts are copied from the build folder to the package folder.

The syntax of `self.copy` inside `package()` is as follows:

```
self.copy(pattern, dst="", src="", keep_path=True, links=False, symlinks=None,␣
↪excludes=None, ignore_case=False)
```

Parameters:

- **pattern** (Required): A pattern following fnmatch syntax of the files you want to copy, from the build to the package folders. Typically something like `*.lib` or `*.h`.
- **src** (Optional, Defaulted to `""`): The folder where you want to search the files in the build folder. If you know that your libraries when you build your package will be in *build/lib*, you will typically use `build/lib` in this parameter. Leaving it empty means the root build folder in local cache.
- **dst** (Optional, Defaulted to `""`): Destination folder in the package. They will typically be `include` for headers, `lib` for libraries and so on, though you can use any convention you like. Leaving it empty means the root package folder in local cache.
- **keep_path** (Optional, Defaulted to `True`): Means if you want to keep the relative path when you copy the files from the **src** folder to the **dst** one. Typically headers are packaged with relative path.
- **symlinks** (Optional, Defaulted to `None`): Set it to True to activate symlink copying, like typical lib.so->lib.so.9.
- **excludes** (Optional, Defaulted to `None`): Single pattern or a tuple of patterns to be excluded from the copy. If a file matches both the include and the exclude pattern, it will be excluded.
- **ignore_case** (Optional, Defaulted to `False`): If enabled, it will do a case-insensitive pattern matching.

For example:

```
self.copy("*.h", "include", "build/include") #keep_path default is True
```

The final path in the package will be: `include/mylib/path/header.h`, and as the *include* is usually added to the path, the includes will be in the form: `#include "mylib/path/header.h"` which is something desired.

`keep_path=False` is something typically desired for libraries, both static and dynamic. Some compilers as MSVC, put them in paths as *Debug/x64/MyLib/Mylib.lib*. Using this option, we could write:

```
self.copy("*.lib", "lib", "", keep_path=False)
```

And it will copy the lib to the package folder *lib/Mylib.lib*, which can be linked easily.

Note: If you are using CMake and you have an install target defined in your CMakeLists.txt, you might be able to reuse it for this `package()` method. Please check *How to reuse cmake install for package() method*.

The `package()` method will be called twice if the attribute `no_copy_source` is defined and `True`. One will copy from the *source* folder (typically packaging the headers and other data files), and the other will copy from the *build* folder, packaging the libraries and other binary artifacts. Also, when the local **`conan package`** command is issued with **`--source-folder`** and **`--build-folder`**, it will execute two times, one in each folder, in the same way.

package_info()

cpp_info

Each package has to specify certain build information for its consumers. This can be done in the `cpp_info` attribute within the `package_info()` method.

The `cpp_info` attribute has the following properties you can assign/append to:

```
self.cpp_info.includedirs = ['include']  # Ordered list of include paths
self.cpp_info.libs = []  # The libs to link against
self.cpp_info.libdirs = ['lib']  # Directories where libraries can be found
self.cpp_info.resdirs = ['res']  # Directories where resources, data, etc can be found
self.cpp_info.bindirs = ['bin']  # Directories where executables and shared libs can␣
↪be found
self.cpp_info.defines = []  # preprocessor definitions
self.cpp_info.cflags = []  # pure C flags
self.cpp_info.cppflags = []  # C++ compilation flags
self.cpp_info.sharedlinkflags = []  # linker flags
self.cpp_info.exelinkflags = []  # linker flags
```

- **includedirs**: List of relative paths (starting from the package root) of directories where headers can be found. By default it is initialized to `['include']`, and it is rarely changed.
- **libs**: Ordered list of libs the client should link against. Empty by default, it is common that different configurations produce different library names. For example:

```
def package_info(self):
    if not self.settings.os == "Windows":
        self.cpp_info.libs = ["libzmq-static.a"] if self.options.static else ["libzmq.
↪so"]
    else:
        ...
```

- **libdirs**: List of relative paths (starting from the package root) of directories in which to find library object binaries (*.lib, *.a, *.so, *.dylib). By default it is initialized to `['lib']`, and it is rarely changed.
- **resdirs**: List of relative paths (starting from the package root) of directories in which to find resource files (images, xml, etc). By default it is initialized to `['res']`, and it is rarely changed.
- **bindirs**: List of relative paths (starting from the package root) of directories in which to find library runtime binaries (like Windows .dlls). By default it is initialized to `['bin']`, and it is rarely changed.
- **defines**: Ordered list of preprocessor directives. It is common that the consumers have to specify some sort of defines in some cases, so that including the library headers matches the binaries:
- **cflags**, **cppflags**, **sharedlinkflags**, **exelinkflags**: List of flags that the consumer should activate for proper behavior. Usage of C++11 could be configured here, for example, although it is true that the consumer may want to do some flag processing to check if different dependencies are setting incompatible flags (c++11 after c++14).

```
if self.options.static:
    if self.settings.compiler == "Visual Studio":
        self.cpp_info.libs.append("ws2_32")
    self.cpp_info.defines = ["ZMQ_STATIC"]

    if not self.settings.os == "Windows":
        self.cpp_info.cppflags = ["-pthread"]
```

If your recipe has requirements, you can access to your requirements `cpp_info` as well using the `deps_cpp_info` object.

```
class OtherConan(ConanFile):
    name = "OtherLib"
    version = "1.0"
    requires = "MyLib/1.6.0@conan/stable"

    def build(self):
        self.output.warn(self.deps_cpp_info["MyLib"].libdirs)
```

Note: Please take into account that defining `self.cpp_info.bindirs` directories, does not have any effect on system paths, PATH environment variable, nor will be directly accessible by consumers. `self.cpp_info` information is translated to build-systems information via generators, for example for CMake, it will be a variable in `conanbuildinfo.cmake`. If you want a package to make accessible its executables to its consumers, you have to specify it with `self.env_info` as described in *env_info*.

env_info

Each package can also define some environment variables that the package needs to be reused. It's specially useful for *installer packages*, to set the path with the "bin" folder of the packaged application. This can be done in the `env_info` attribute within the `package_info()` method.

```
self.env_info.path.append("ANOTHER VALUE") # Append "ANOTHER VALUE" to the path␣
↪variable
self.env_info.othervar = "OTHER VALUE" # Assign "OTHER VALUE" to the othervar variable
self.env_info.thirdvar.append("some value") # Every variable can be set or appended a␣
↪new value
```

One of the most typical usages for the PATH environment variable, would be to add the current binary package directories to the path, so consumers can use those executables easily:

```
# assuming the binaries are in the "bin" subfolder
self.env_info.PATH.append(os.path.join(self.package_folder, "bin")
```

The *virtualenv* generator will use the self.env_info variables to prepare a script to activate/deactive a virtual environment.

They will be automatically applied before calling the consumer *conanfile.py* methods `source()`, `build()`, `package()` and `imports()`.

If your recipe has requirements, you can access to your requirements `env_info` as well using the `deps_env_info` object.

```
class OtherConan(ConanFile):
    name = "OtherLib"
    version = "1.0"
    requires = "MyLib/1.6.0@conan/stable"

    def build(self):
        self.output.warn(self.deps_env_info["MyLib"].othervar)
```

user_info

If you need to declare custom variables not related with C/C++ (`cpp_info`) and the variables are not environment variables (`env_info`), you can use the `self.user_info` object.

Currently only the `cmake`, `cmake_multi` and `txt` generators supports `user_info` variables.

```
class MyLibConan(ConanFile):
    name = "MyLib"
    version = "1.6.0"

    # ...

    def package_info(self):
        self.user_info.var1 = 2
```

For the example above, in the `cmake` and `cmake_multi` generators, a variable `CONAN_USER_MYLIB_var1` will be declared. If your recipe has requirements, you can access to your requirements `user_info` using the `deps_user_info` object.

```
class OtherConan(ConanFile):
    name = "OtherLib"
    version = "1.0"
    requires = "MyLib/1.6.0@conan/stable"

    def build(self):
        self.out.warn(self.deps_user_info["MyLib"].var1)
```

configure(), config_options()

If the package options and settings are related, and you want to configure either, you can do so in the `configure()` and `config_options()` methods.

```
class MyLibConan(ConanFile):
    name = "MyLib"
    version = "2.5"
    settings = "os", "compiler", "build_type", "arch"
    options = {"static": [True, False],
               "header_only": [True False]}

    def configure(self):
        # If header only, the compiler, etc, does not affect the package!
        if self.options.header_only:
            self.settings.clear()
            self.options.remove("static")
```

The package has 2 options set, to be compiled as a static (as opposed to shared) library, and also not to involve any builds, because header-only libraries will be used. In this case, the settings that would affect a normal build, and even the other option (static vs shared) do not make sense, so we just clear them. That means, if someone consumes MyLib with the `header_only=True` option, the package downloaded and used will be the same, irrespective of the OS, compiler or architecture the consumer is building with.

You can also restrict the settings used deleting any specific one. For example, it is quite common for C libraries to delete the `libcxx` as your library does not depend on any C++ standard library:

```
def configure(self):
    del self.settings.compiler.libcxx
```

The most typical usage would be the one with `configure()` while `config_options()` should be used more sparingly. `config_options()` is used to configure or constraint the available options in a package, **before** they are given a value. So when a value is tried to be assigned it will raise an error. For example, let's suppose that a certain package library cannot be built as shared library in Windows, it can be done:

```
def config_options(self):
    if self.settings.os == "Windows":
        del self.options.shared
```

This will be executed before the actual assignment of `options` (then, such `options` values cannot be used inside this function), so the command **`conan install -o Pkg:shared=True`** will raise an exception in Windows saying that `shared` is not an option for such package.

requirements()

Besides the `requires` field, more advanced requirement logic can be defined in the `requirements()` optional method, using for example values from the package `settings` or `options`:

```
def requirements(self):
    if self.options.myoption:
        self.requires("zlib/1.2@drl/testing")
    else:
        self.requires("opencv/2.2@drl/stable")
```

This is a powerful mechanism for handling **conditional dependencies**.

When you are inside the method, each call to `self.requires()` will add the corresponding requirement to the current list of requirements. It also has optional parameters that allow defining the special cases, as is shown below:

```
def requirements(self):
    self.requires("zlib/1.2@drl/testing", private=True, override=False)
```

`self.requires()` parameters:

- **override** (Optional, Defaulted to `False`): True means that this is not an actual requirement, but something to be passed upstream and override possible existing values.
- **private** (Optional, Defaulted to `False`): True means that this requirement will be somewhat embedded (like a static lib linked into a shared lib), so it is not required to link.

build_requirements()

Build requirements are requirements that are only installed and used when the package is built from sources. If there is an existing pre-compiled binary, then the build requirements for this package will not be retrieved.

This method is useful for defining conditional build requirements, for example:

```
class MyPkg(ConanFile):

    def build_requirements(self):
        if self.settings.os == "Windows":
            self.build_requires("ToolWin/0.1@user/stable")
```

See also:

Build requirements

system_requirements()

It is possible to install system-wide packages from conan. Just add a `system_requirements()` method to your conanfile and specify what you need there.

For a special use case you can use also `conans.tools.os_info` object to detect the operating system, version and distribution (linux):

- `os_info.is_linux`: True if Linux.
- `os_info.is_windows`: True if Windows.
- `os_info.is_macos`: True if OSx.
- `os_info.is_freebsd`: True if FreeBSD.
- `os_info.is_solaris`: True if SunOS.
- `os_info.os_version`: OS version.
- `os_info.os_version_name`: Common name of the OS (Windows 7, Mountain Lion, Wheezy...).
- `os_info.linux_distro`: Linux distribution name (None if not Linux).
- `os_info.bash_path`: Returns the absolute path to a bash in the system.
- `os_info.uname(options=None)`: Runs the "uname" command and returns the ouput. You can pass arguments with the *options* parameter.
- `os_info.detect_windows_subsystem()`: Returns "MSYS", "MSYS2", "CYGWIN" or "WSL" if any of these Windows subsystems are detected.

You can also use `SystemPackageTool` class, that will automatically invoke the right system package tool: **apt**, **yum**, **pkg**, **pkgutil**, **brew** and **pacman** depending on the system we are running.

```
from conans.tools import os_info, SystemPackageTool

def system_requirements(self):
    pack_name = None
    if os_info.linux_distro == "ubuntu":
        if os_info.os_version > "12":
            pack_name = "package_name_in_ubuntu_10"
        else:
            pack_name = "package_name_in_ubuntu_12"
    elif os_info.linux_distro == "fedora" or os_info.linux_distro == "centos":
        pack_name = "package_name_in_fedora_and_centos"
    elif os_info.is_macos:
        pack_name = "package_name_in_macos"
    elif os_info.is_freebsd:
        pack_name = "package_name_in_freebsd"
    elif os_info.is_solaris:
        pack_name = "package_name_in_solaris"

    if pack_name:
        installer = SystemPackageTool()
        installer.install(pack_name) # Install the package, will update the package␣
↪database if pack_name isn't already installed
```

On Windows, there is no standard package manager, however **choco** can be invoked as an optional:

```
from conans.tools import os_info, SystemPackageTool, ChocolateyTool

def system_requirements(self):
    if os_info.is_windows:
        pack_name = "package_name_in_windows"
        installer = SystemPackageTool(tool=ChocolateyTool()) # Invoke choco package␣
↪manager to install the package
        installer.install(pack_name)
```

SystemPackageTool

```
def SystemPackageTool(tool=None)
```

Available tool classes: **AptTool**, **YumTool**, **BrewTool**, **PkgTool**, **PkgUtilTool**, **ChocolateyTool**, **PacManTool**.

Methods:

- **update()**: Updates the system package manager database. It's called automatically from the `install()` method by default.
- **install(packages, update=True, force=False)**: Installs the `packages` (could be a list or a string). If `update` is True it will execute `update()` first if it's needed. The packages won't be installed if they are already installed at least of `force` parameter is set to True. If `packages` is a list the first available package will be picked (short-circuit like logical **or**).

The use of `sudo` in the internals of the `install()` and `update()` methods is controlled by the `CONAN_SYSREQUIRES_SUDO` environment variable, so if the users don't need sudo permissions, it is easy to opt-in/out.

Conan will keep track of the execution of this method, so that it is not invoked again and again at every Conan command. The execution is done per package, since some packages of the same library might have different system dependencies. If you are sure that all your binary packages have the same system requirements, just add the following line to your method:

```
def system_requirements(self):
    self.global_system_requirements=True
    if ...
```

imports()

Importing files copies files from the local store to your project. This feature is handy for copying shared libraries (*dylib* in Mac, *dll* in Win) to the directory of your executable, so that you don't have to mess with your PATH to run them. But there are other use cases:

- Copy an executable to your project, so that it can be easily run. A good example is the **Google's protobuf** code generator.
- Copy package data to your project, like configuration, images, sounds... A good example is the **OpenCV** demo, in which face detection XML pattern files are required.

Importing files is also very convenient in order to redistribute your application, as many times you will just have to bundle your project's bin folder.

A typical `imports()` method for shared libs could be:

```
def imports(self):
    self.copy("*.dll", "", "bin")
    self.copy("*.dylib", "", "lib")
```

The `self.copy()` method inside `imports()` supports the following arguments:

```
def copy(pattern, dst="", src="", root_package=None, folder=False, ignore_case=False,␣
↪excludes=None, keep_path=True)
```

Parameters:

- **pattern** (Required): An fnmatch file pattern of the files that should be copied.
- **dst** (Optional, Defaulted to `""`): Destination local folder, with reference to current directory, to which the files will be copied.
- **src** (Optional, Defaulted to `""`): Source folder in which those files will be searched. This folder will be stripped from the dst parameter. Eg.: lib/Debug/x86
- **root_package** (Optional, Defaulted to *all packages in deps*): An fnmatch pattern of the package name ("OpenCV", "Boost") from which files will be copied.
- **folder** (Optional, Defaulted to `False`): If enabled, it will copy the files from the local cache to a subfolder named as the package containing the files. Useful to avoid conflicting imports of files with the same name (e.g. License).
- **ignore_case** (Optional, Defaulted to `False`): If enabled, it will do a case-insensitive pattern matching.
- **excludes** (Optional, Defaulted to `None`): Allows defining a list of patterns (even a single pattern) to be excluded from the copy, even if they match the main `pattern`.
- **keep_path** (Optional, Defaulted to `True`): Means if you want to keep the relative path when you copy the files from the **src** folder to the **dst** one. Useful to ignore (`keep_path=False`) path of *library.dll* files in the package it is imported from.

Example to collect license files from dependencies:

```
def imports(self):
    self.copy("license*", dst="licenses", folder=True, ignore_case=True)
```

If you want to be able to customize the output user directory to work with both the `cmake` and `cmake_multi` generators, then you can do:

```
def imports(self):
    dest = os.getenv("CONAN_IMPORT_PATH", "bin")
    self.copy("*.dll", dst=dest, src="bin")
    self.copy("*.dylib*", dst=dest, src="lib")
```

And then use, for example: **`conan install . -e CONAN_IMPORT_PATH=Release -g cmake_multi`**

When a conanfile recipe has an `imports()` method and it builds from sources, it will do the following:

- Before running `build()` it will execute `imports()` in the build folder, copying dependencies artifacts
- Run the `build()` method, which could use such imported binaries.
- Remove the copied (imported) artifacts after `build()` is finished.

You can use the *keep_imports* attribute to keep the imported artifacts, and maybe *repackage* them.

package_id()

Creates a unique ID for the package. Default package ID is calculated using `settings`, `options` and `requires` properties. When a package creator specifies the values for any of thoses properties, it is telling that any value change will require a different binary package.

However, sometimes a package creator would need to alter the default behavior, for example, to have only one binary package for several different compiler versions. In that case you can set a custom `self.info` object implementing this method and the package ID will be computed with the given information:

```
def package_id(self):
    v = Version(str(self.settings.compiler.version))
    if self.settings.compiler == "gcc" and (v >= "4.5" and v < "5.0"):
        self.info.settings.compiler.version = "GCC 4 between 4.5 and 5.0"
```

Please, check the section *Define package ABI compatibility* to get more details.

self.info

This `self.info` object stores the information that will be used to compute the package ID.

This object can be manipulated to reflect the information you want in the computation of the package ID. For example, you can delete any setting or option:

```
def package_id(self):
    del self.info.settings.compiler
    del self.info.options.shared
```

self.info.header_only()

The package will always be the same, irrespective of the OS, compiler or architecture the consumer is building with.

```
def package_id(self):
    self.info.header_only()
```

self.info.vs_toolset_compatible() / self.info.vs_toolset_incompatible()

By default (`vs_toolset_compatible()` mode) Conan will generate the same binary package when the compiler is Visual Studio and the `compiler.toolset` matches the specified `compiler.version`. For example, if we install some packages specifying the following settings:

```
def package_id(self):
    self.info.vs_toolset_compatible()
    # self.info.vs_toolset_incompatible()
```

```
compiler="Visual Studio"
compiler.version=14
```

And then we install again specifying these settings:

```
compiler="Visual Studio"
compiler.version=15
compiler.toolset=v140
```

The compiler version is different, but Conan will not install a different package, because the used `toolchain` in both cases are considered the same. You can deactivate this default behavior using calling `self.info.vs_toolset_incompatible()`.

This is the relation of Visual Studio versions and the compatible toolchain:

Visual Studio Version	Compatible toolset
15	v141
14	v140
13	v120
12	v120
11	v110
10	v100
9	v90
8	v80

self.info.discard_build_settings() / self.info.include_build_settings()

By default (`discard_build_settings()`) Conan will generate the same binary when you change the `os_build` or `arch_build` when the `os` and `arch` are declared respectively. This is because `os_build` represent the machine running Conan, so, for the consumer, the only setting that matters is where the built software will run, not where is running the compilation. The same applies to `arch_build`.

With `self.info.include_build_settings()`, Conan will generate different packages when you change the `os_build` or `arch_build`.

```
def package_id(self):
    self.info.discard_build_settings()
    # self.info.include_build_settings()
```

self.info.default_std_matching() / self.info.default_std_non_matching()

By default (`default_std_matching()`) Conan will detect the default C++ standard of your compiler to not generate different binary packages.

For example, you already built some `gcc > 6.1` packages, where the default std is `gnu14`. If you introduce the `cppstd` setting in your recipes and specify the `gnu14` value, Conan won't generate new packages, because it was already the default of your compiler.

With `self.info.default_std_non_matching()`, Conan will generate different packages when you specify the `cppstd` even if it matches with the default of the compiler being used:

```
def package_id(self):
    self.info.default_std_non_matching()
    # self.info.default_std_matching()
```

build_id()

In the general case, there is one build folder for each binary package, with the exact same hash/ID of the package. However this behavior can be changed, there are a couple of scenarios that this might be interesting:

- You have a build script that generates several different configurations at once, like both debug and release artifacts, but you actually want to package and consume them separately. Same for different architectures or any other setting.
- You build just one configuration (like release), but you want to create different binary packages for different consuming cases. For example, if you have created tests for the library in the build step, you might want to create two packages: one just containing the library for general usage, and another one also containing the tests. First package could be used as a reference and the other one as a tool to debug errors.

In both cases, if using different settings, the system will build twice (or more times) the same binaries, just to produce a different final binary package. With the `build_id()` method this logic can be changed. `build_id()` will create a new package ID/hash for the build folder, and you can define the logic you want in it. For example:

```
settings = "os", "compiler", "arch", "build_type"

def build_id(self):
    self.info_build.settings.build_type = "Any"
```

So this recipe will generate a final different package for each debug/release configuration. But as the `build_id()` will generate the same ID for any `build_type`, then just one folder and one build will be done. Such build should build both debug and release artifacts, and then the `package()` method should package them accordingly to the `self.settings.build_type` value. Different builds will still be executed if using different compilers or architectures. This method is basically an optimization of build time, avoiding multiple re-builds.

Other information like custom package options can also be changed:

```
def build_id(self):
    self.info_build.options.myoption = 'MyValue' # any value possible
    self.info_build.options.fullsource = 'Always'
```

If the `build_id()` method does not modify the `build_id`, and produce a different one than the `package_id`, then the standard behavior will be applied. Consider the following:

```
settings = "os", "compiler", "arch", "build_type"

def build_id(self):
    if self.settings.os == "Windows":
        self.info_build.settings.build_type = "Any"
```

This will only produce a build ID different if the package is for Windows. So the behavior in any other OS will be the standard one, as if the `build_id()` method was not defined: the build folder will be wiped at each **`conan create`** command and a clean build will be done.

deploy()

This method can be used in a *conanfile.py* to install in the system or user folder artifacts from packages.

```
def deploy(self):
    self.copy("*.exe")  # copy from current package
    self.copy_deps("*.dll") # copy from dependencies
```

Where:

- `self.copy()` is the `self.copy()` method executed inside *package() method*.
- `self.copy_deps()` is the same as `self.copy()` method inside *imports() method*.

Both methods allow the definition of absolute paths (to install in the system), in the `dst` argument. By default, the `dst` destionation folder will be the current one.

The `deploy()` method is designed to work on a package that is installed directly from its reference, as:

```
$ conan install Pkg/0.1@user/channel
> ...
> Pkg/0.1@user/testing deploy(): Copied 1 '.dll' files: mylib.dll
> Pkg/0.1@user/testing deploy(): Copied 1 '.exe' files: myexe.exe
```

All other packages and dependencies, even transitive dependencies of "Pkg/0.1@user/testing" will not be deployed, it is the responsibility of the installed package to deploy what it needs from its dependencies.

13.3.3 Output and Running

Output contents

Use the *self.output* to print contents to the output.

```
self.output.success("This is a good, should be green")
self.output.info("This is a neutral, should be white")
self.output.warn("This is a warning, should be yellow")
self.output.error("Error, should be red")
self.output.rewrite_line("for progress bars, issues a cr")
```

Check the source code. You might be able to produce different outputs with different colors.

Running commands

```
run(self, command, output=True, cwd=None, win_bash=False, subsystem=None, msys_
↪mingw=True):
```

`self.run()` is a helper to run system commands and throw exceptions when errors occur, so that command errors are do not pass unnoticed. It is just a wrapper for `os.system()`

Optional parameters:

- **output (Optional, Defaulted to `True`) When True it will write in stdout.** You can pass any stream that accepts a `write` method like a `six.StringIO()`:

```
from six import StringIO  # Python 2 and 3 compatible
mybuf = StringIO()
self.run("mycommand", output=mybuf)
self.output.warn(mybuf.getvalue())
```

- **cwd** (Optional, Defaulted to `.` current directory): Current directory to run the command.
- **win_bash** (Optional, Defaulted to `False`): When True, it will run the configure/make commands inside a bash.
- **subsystem** (Optional, Defaulted to `None` will autodetect the subsystem). Used to escape the command according to the specified subsystem.
- **msys_mingw** (Optional, Defaulted to `True`) If the specified subsystem is MSYS2, will start it in MinGW mode (native windows development).

13.4 Generators

You can specify a generator in:

- The **[generators]** section from *conanfile.txt*
- The **generators** attribute in *conanfile.py*

Available generators:

13.4.1 *cmake*

This is the reference page for `cmake` generator. Go to *Integrations/CMake* if you want to learn how to integrate your project or recipes with CMake.

It generates a file named `conanbuildinfo.cmake` and declares some variables and methods

Variables in conanbuildinfo.cmake

- **Package declared vars**

For each requirement `conanbuildinfo.cmake` file declares the following variables. `XXX` is the name of the require in uppercase. e.k "ZLIB" for `zlib/1.2.8@lasote/stable` requirement:

NAME	VALUE
CONAN_XXX_ROOT	Abs path to root package folder.
CONAN_INCLUDE_DIRS_XXX	Header's folders
CONAN_LIB_DIRS_XXX	Library folders (default {CONAN_XXX_ROOT}/lib)
CONAN_BIN_DIRS_XXX	Binary folders (default {CONAN_XXX_ROOT}/bin)
CONAN_LIBS_XXX	Library names to link
CONAN_DEFINES_XXX	Library defines
CONAN_COMPILE_DEFINITIONS_XXX	Compile definitions
CONAN_CXX_FLAGS_XXX	CXX flags
CONAN_SHARED_LINK_FLAGS_XXX	Shared link flags
CONAN_C_FLAGS_XXX	C flags

- **Global declared vars**

Conan also declares some global variables with the aggregated values of all our requirements. The values are ordered in the right order according to the dependency tree.

NAME	VALUE
CONAN_INCLUDE_DIRS	Aggregated header's folders
CONAN_LIB_DIRS	Aggregated library folders
CONAN_BIN_DIRS	Aggregated binary folders
CONAN_LIBS	Aggregated library names to link
CONAN_DEFINES	Aggregated library defines
CONAN_COMPILE_DEFINITIONS	Aggregated compile definitions
CONAN_CXX_FLAGS	Aggregated CXX flags
CONAN_SHARED_LINK_FLAGS	Aggregated Shared link flags
CONAN_C_FLAGS	Aggregated C flags

- **Variables from user_info**

If any of the requirements is filling the *user_info* object in the *package_info* method a set of variables will be declared following this naming:

NAME	VALUE
CONAN_USER_XXXX_YYYY	User declared value

XXXX is the name of the requirement in uppercase and YYYY the variable name. e.j:

```
class MyLibConan(ConanFile):
    name = "MyLib"
    version = "1.6.0"

    # ...

    def package_info(self):
        self.user_info.var1 = 2
```

When other library requires MyLib and uses the cmake generator:

conanbuildinfo.cmake:

```
# ...
set(CONAN_USER_MYLIB_var1 "2")
```

Methods available in conanbuildinfo.cmake

conan_basic_setup

Setup all the CMake vars according to our settings, by default with the global approach (no targets).

parameters: You can combine several parameters to the conan_basic_setup macro. e.j: conan_basic_setup(TARGETS KEEP_RPATHS)

- TARGETS: Setup all the CMake vars by target (only CMake > 3.1.2)
- NO_OUTPUT_DIRS: Do not adjust the output directories
- KEEP_RPATHS: Do not adjust the CMAKE_SKIP_RPATH variable in OSX

conan_target_link_libraries

Helper to link all libraries to a specified target.

Other optional methods

There are other methods automatically called by conan_basic_setup() but you can use them directly:

NAME	DESCRIPTION
conan_check_compiler()	Checks that your compiler matches with the declared in the settings
conan_output_dirs_setup()	Adjust the bin/ and lib/ output directories
conan_set_find_library_paths()	Set CMAKE_INCLUDE_PATH and CMAKE_INCLUDE_PATH
conan_global_flags()	Set include_directories, link_directories, link_directories, flags
conan_define_targets()	Define the targets (target flags instead of global flags)
conan_set_rpath()	Set CMAKE_SKIP_RPATH=1 if APPLE
conan_set_vs_runtime()	Adjust the runtime flags (/MD /MDd /MT /MTd)
conan_set_libcxx(TARGETS)	Adjust the standard library flags (libstdc++, libc++, libstdc++11)
conan_set_find_paths()	Adjust CMAKE_MODULE_PATH and CMAKE_PREFIX_PATH

Targets generated by conanbuildinfo.cmake

If you use `conan_basic_setup(TARGETS)`, then some cmake targets will be generated (this only works for CMake > 3.1.2)

These targets are:

- A `CONAN_PKG::PkgName` target per package in the dependency graph. This is an `IMPORTED INTERFACE` target. IMPORTED because it is external, a pre-compiled library. INTERFACE, because it doesn't necessarily match a library, it could be a header-only library, or the package could even contain several libraries. It contains all the properties (include paths, compile flags, etc) that are defined in the `package_info()` method of the package.
- Inside each package a `CONAN_LIB::PkgName_LibName` target will be generated for each library. Its type is `IMPORTED UNKNOWN`, its mainly purpose is to provide a correct link order. Their only properties are the location and the dependencies
- A `CONAN_PKG` depends on every `CONAN_LIB` that belongs to it, and to its direct public dependencies (i.e. other `CONAN_PKG` targets from its `requires`)
- Each `CONAN_LIB` depends on the direct public dependencies `CONAN_PKG` targets of its container package. This guarantees correct link order.

13.4.2 *cmake_multi*

This is the reference page for `cmake_multi` generator. Go to *Integrations/CMake* if you want to learn how to integrate your project or recipes with CMake.

Usage

```
$ conan install -g cmake_multi -s build_type=Release ...
$ conan install -g cmake_multi -s build_type=Debug ...
```

These commands will generate 3 files:

- `conanbuildinfo_release.cmake`: Variables adjusted only for build_type Release
- `conanbuildinfo_debug.cmake`: Variables adjusted only for build_type Debug
- `conanbuildinfo_multi.cmake`: Which includes the other two, and enables its use

Variables in conanbuildinfo_release.cmake

Same as *conanbuildinfo.cmake* with suffix _RELEASE

Variables in conanbuildinfo_debug.cmake

Same as *conanbuildinfo.cmake* with suffix _DEBUG

Available Methods

Same as *conanbuildinfo.cmake*

13.4.3 visual_studio

This is the reference page for `visual_studio` generator. Go to *Integrations/Visual Studio* if you want to learn how to integrate your project or recipes with Visual Studio.

Generates a file named `conanbuildinfo.props` containing an XML that can be imported to your *Visual Studio* project.

Generated xml structure:

```
<?xml version="1.0" encoding="utf-8"?>
<Project ToolsVersion="4.0" xmlns="http://schemas.microsoft.com/developer/msbuild/2003
↪">
  <ImportGroup Label="PropertySheets" />
  <PropertyGroup Label="UserMacros" />
  <PropertyGroup Label="Conan-RootDirs">
    <Conan-Lib1-Root>{PACKAGE LIB1 FOLDER}</Conan-Poco-Root>
    <Conan-Lib2-Root>{PACKAGE LIB2 FOLDER}</Conan-Poco-Root>
    ...
  </PropertyGroup>
  <PropertyGroup Label="ConanVariables">
    <ConanBinaryDirectories>{CONAN BINARY DIRECTORIES LIST}</ConanBinaryDirectories>
    <ConanResourceDirectories>{CONAN RESOURCE DIRECTORIES LIST}</
↪ConanResourceDirectories>
  </PropertyGroup>
  <PropertyGroup>
    <LocalDebuggerEnvironment>PATH=%PATH%;{CONAN BINARY DIRECTORIES LIST}</
↪LocalDebuggerEnvironment>
    <DebuggerFlavor>WindowsLocalDebugger</DebuggerFlavor>
  </PropertyGroup>
  <ItemDefinitionGroup>
    <ClCompile>
      <AdditionalIncludeDirectories>{CONAN INCLUDE DIRECTORIES LIST}
↪%(AdditionalIncludeDirectories)</AdditionalIncludeDirectories>
      <PreprocessorDefinitions>{CONAN DEFINITIONS}%(PreprocessorDefinitions)</
↪PreprocessorDefinitions>
      <AdditionalOptions> %(AdditionalOptions)</AdditionalOptions>
    </ClCompile>
    <Link>
      <AdditionalLibraryDirectories>{CONAN LIB DIRECTORIES LIST}
↪%(AdditionalLibraryDirectories)</AdditionalLibraryDirectories>
      <AdditionalDependencies>{CONAN LIBS LIST}</AdditionalDependencies>
      <AdditionalOptions> %(AdditionalOptions)</AdditionalOptions>
```

(continues on next page)

(continued from previous page)

```
    </Link>
  </ItemDefinitionGroup>
  <ItemGroup />
</Project>
```

Note that for single-configuration packages, which is the most typical, conan install Debug/Release, 32/64bits, packages separately. So a different property sheet will be generated for each configuration. The process could be:

Given for example a `conanfile.txt` like:

```
[requires]
Pkg/0.1@user/channel

[generators]
visual_studio
```

And assuming that binary packages exist for `Pkg/0.1@user/channel`, we could do:

```
$ mkdir debug32 && cd debug32
$ conan install .. -s compiler="Visual Studio" -s compiler.version=15 -s arch=x86 -s␣
↪build_type=Debug
$ cd ..
$ mkdir debug64 && cd debug64
$ conan install .. -s compiler="Visual Studio" -s compiler.version=15 -s arch=x86_64 -
↪s build_type=Debug
$ cd ..
$ mkdir release32 && cd release32
$ conan install .. -s compiler="Visual Studio" -s compiler.version=15 -s arch=x86 -s␣
↪build_type=Release
$ cd ..
$ mkdir release64 && cd release64
$ conan install .. -s compiler="Visual Studio" -s compiler.version=15 -s arch=x86_64 -
↪s build_type=Release
...
# Now go to VS 2017 Property Manager, load the respective sheet into each␣
↪configuration
```

The above process can be simplified using profiles (assuming you have created the respective profiles), and you can also specify the generators in the command line:

```
$ conan install .. -pr=vs15release64 -g visual_studio
...
```

13.4.4 *visual_studio_multi*

This is the reference page for `visual_studio_multi` generator. Go to *Integrations/Visual Studio* if you want to learn how to integrate your project or recipes with Visual Studio.

Usage

```
$ conan install . -g visual_studio_multi -s arch=x86 -s build_type=Debug
$ conan install . -g visual_studio_multi -s arch=x86_64 -s build_type=Debug
$ conan install . -g visual_studio_multi -s arch=x86 -s build_type=Release
$ conan install . -g visual_studio_multi -s arch=x86_64 -s build_type=Release
```

These commands will generate 5 files for each compiler version:

- `conanbuildinfo_multi.props`: All properties
- `conanbuildinfo_release_x64_v141.props.props`: Variables for release/64bits/VS2015 (toolset v141)
- `conanbuildinfo_debug_x64_v141.props.props`: Variables for debug/64bits/VS2015 (toolset v141)
- `conanbuildinfo_release_win32_v141.props.props`: Variables for release/32bits/VS2015 (toolset v141)
- `conanbuildinfo_debug_win32_v141.props.props`: Variables for debug/32bits/VS2015 (toolset v141)

You can now load `conanbuildinfo_multi.props` in your Visual Studio IDE property manager, and all configurations will be loaded at once.

Each one of the configurations will have the format and information defined in *the visual_studio generator*

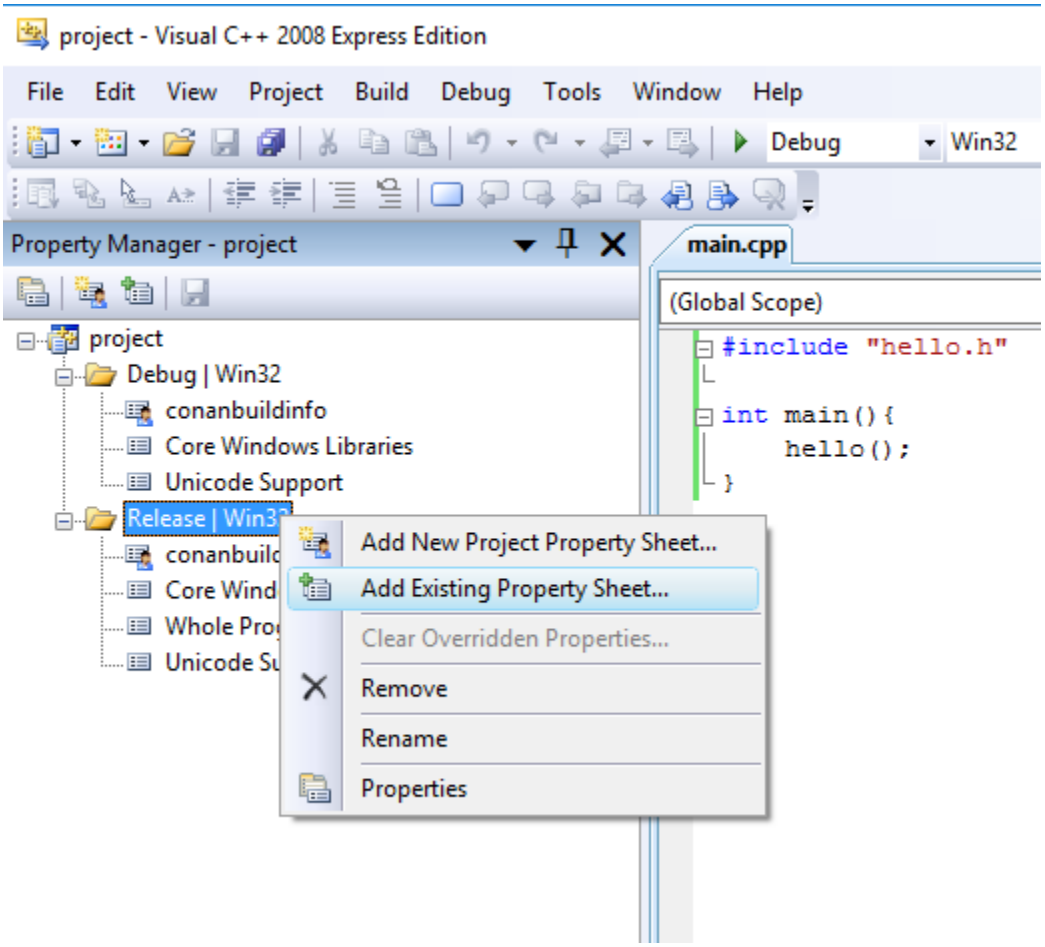
13.4.5 visual_studio_legacy

Generates a file named `conanbuildinfo.vsprops` containing an XML that can be imported to your *Visual Studio 2008* project. Note that the format of this file is different and incompatible with the `conanbuildinfo.props` file generated with the `visual_studio` generator for newer VS.

Generated xml structure:

```
<?xml version="1.0" encoding="Windows-1252"?>
<VisualStudioPropertySheet
    ProjectType="Visual C++"
    Version="8.00"
    Name="conanbuildinfo"
    >
    <Tool
        Name="VCCLCompilerTool"
        AdditionalOptions="{compiler_flags}"
        AdditionalIncludeDirectories="{include_dirs}"
        PreprocessorDefinitions="{definitions}"
    />
    <Tool
        Name="VCLinkerTool"
        AdditionalOptions="{linker_flags}"
        AdditionalDependencies="{libs}"
        AdditionalLibraryDirectories="{lib_dirs}"
    />
</VisualStudioPropertySheet>
```

This file can be loaded from the Menu->View->PropertyManager window, selecting "Add Existing Property Sheet" for the desired configuration.



Note that for single-configuration packages, which is the most typical, conan install Debug and Release packages separately. So a different property sheet will be generated for each configuration. The process could be:

Given for example a conanfile.txt like:

```
[requires]
Pkg/0.1@user/channel

[generators]
visual_studio_legacy
```

And assuming that binary packages exist for `Pkg/0.1@user/channel`, we could do:

```
$ mkdir debug && cd debug
$ conan install .. -s compiler="Visual Studio" -s compiler.version=9 -s arch=x86 -s␣
↪build_type=Debug
$ cd ..
$ mkdir release && cd release
$ conan install .. -s compiler="Visual Studio" -s compiler.version=9 -s arch=x86 -s␣
↪build_type=Release
# Now go to VS 2008 Property Manager, load the respective sheet into each␣
↪configuration
```

The above process can be simplified using profiles (assuming you have created "vs9release" profile), and you can also specify the generators in the command line:

```
$ conan install .. -pr=vs9release -g visual_studio_legacy
```

13.4.6 xcode

This is the reference page for `xcode` generator. Go to *Integrations/Xcode* if you want to learn how to integrate your project or recipes with Xcode.

The **xcode** generator creates a file named `conanbuildinfo.xcconfig` that can be imported to your *Xcode* project.

The file declare these variables:

VARIABLE	VALUE
HEADER_SEARCH_PATHS	The requirements *include dirs*
LIBRARY_SEARCH_PATHS	The requirements *lib dirs*
OTHER_LDFLAGS	*-lXXX* corresponding to library names
GCC_PREPROCESSOR_DEFINITIONS	The requirements definitions
OTHER_CFLAGS	The requirements cflags
OTHER_CPLUSPLUSFLAGS	The requirements cxxflags

13.4.7 compiler_args

This is the reference page for `compiler_args` generator. Go to *Integrations/Compilers on command line* if you want to learn how to integrate your project calling your compiler in the command line.

Generates a file named `conanbuildinfo.args` containing a command line parameters to invoke `gcc`, `clang` or `cl` compiler.

You can use the **compiler_args** generator directly to build simple programs:

gcc/clang:

```
> g++ timer.cpp @conanbuildinfo.args -o bin/timer
```

cl:

```
$ cl /EHsc timer.cpp @conanbuildinfo.args
```

gcc/clang

FLAG	MEANING
-DXXX	Corresponding to requirements *defines*
-IXXX	Corresponding to requirements *include dirs*
-Wl,-rpathXXX	Corresponding to requirements *lib dirs*
-LXXX	Corresponding to requirements *lib dirs*
-lXXX	Corresponding to requirements *libs*
-m64	For x86_64 architecture
-m32	For x86 architecture
-DNDEBUG	For Release builds
-s	For Release builds (only gcc)
-g	For Debug builds
-D_GLIBCXX_USE_CXX11_ABI=0	When setting libcxx == "libstdc++"
-D_GLIBCXX_USE_CXX11_ABI=1	When setting libcxx == "libstdc++11"
Other flags	cppflags, cflags, sharedlinkflags, exelinkflags (applied directly)

cl (Visual Studio)

FLAG	MEANING
/DXXX	Corresponding to requirements *defines*
/IXXX	Corresponding to requirements *include dirs*
/LIBPATH:XX	Corresponding to requirements *lib dirs*
/MT, /MTd, /MD, /MDd	Corresponding to Runtime
-DNDEBUG	For Release builds
/Zi	For Debug builds

You can also use it in a recipe:

```
from conans import ConanFile

class PocoTimerConan(ConanFile):
   settings = "os", "compiler", "build_type", "arch"
   requires = "Poco/1.9.0@pocoproject/stable"
   generators = "compiler_args"
   default_options = "Poco:shared=True", "OpenSSL:shared=True"

   def imports(self):
      self.copy("*.dll", dst="bin", src="bin") # From bin to bin
      self.copy("*.dylib*", dst="bin", src="lib") # From lib to bin

   def build(self):
      self.run("mkdir -p bin")
      command = 'g++ timer.cpp @conanbuildinfo.args -o bin/timer'
      self.run(command)
```

13.4.8 gcc

Deprecated, use *compiler_args* generator instead.

13.4.9 Boost Build

The **boost-build** generator creates a file named `project-root.jam` that can be used with your *Boost Build* build system script.

The generated `project-root.jam` will generate several sections, and an alias `conan-deps` with the sections name:

```
lib ssl :
    : # requirements
    <name>ssl
    <search>/path/to/package/227fb0ea22f4797212e72ba94ea89c7b3fbc2a0c/lib
    : # default-build
    : # usage-requirements
    <include>/path/to/package/227fb0ea22f4797212e72ba94ea89c7b3fbc2a0c/include
    ;

lib crypto :
    : # requirements
    <name>crypto
    <search>/path/to/package/227fb0ea22f4797212e72ba94ea89c7b3fbc2a0c/lib
    : # default-build
    : # usage-requirements
    <include>/path/to/package/227fb0ea22f4797212e72ba94ea89c7b3fbc2a0c/include
    ;

lib z :
    : # requirements
    <name>z
    <search>/path/to/package/8018a4df6e7d2b4630a814fa40c81b85b9182d2b/lib
    : # default-build
    : # usage-requirements
    <include>/path/to/package/8018a4df6e7d2b4630a814fa40c81b85b9182d2b/include
    ;

alias conan-deps :
    ssl
    crypto
    z
;
```

13.4.10 qbs

This is the reference page for `qbs` generator. Go to *Integrations/Qbs* if you want to learn how to integrate your project or recipes with Qbs.

Generates a file named `conanbuildinfo.qbs` that can be used for your qbs builds.

A Product `ConanBasicSetup` contains the aggregated requirement values and also there is N Product declared, one per requirement.

```
import qbs 1.0

Project {
    Product {
        name: "ConanBasicSetup"
        Export {
            Depends { name: "cpp" }
            cpp.includePaths: [{INCLUDE DIRECTORIES REQUIRE 1}, {INCLUDE DIRECTORIES␣
↪REQUIRE 2}]
            cpp.libraryPaths: [{LIB DIRECTORIES REQUIRE 1}, {LIB DIRECTORIES REQUIRE␣
↪2}]
```

(continues on next page)

(continued from previous page)

```
            cpp.systemIncludePaths: [{BIN DIRECTORIES REQUIRE 1}, {BIN DIRECTORIES␣
↪REQUIRE 2}]
            cpp.dynamicLibraries: [{LIB NAMES REQUIRE 1}, {LIB NAMES REQUIRE 2}]
            cpp.defines: []
            cpp.cppFlags: []
            cpp.cFlags: []
            cpp.linkerFlags: []
        }
    }

    Product {
        name: "REQUIRE1"
        Export {
            Depends { name: "cpp" }
            cpp.includePaths: [{INCLUDE DIRECTORIES REQUIRE 1}]
            cpp.libraryPaths: [{LIB DIRECTORIES REQUIRE 1}]
            cpp.systemIncludePaths: [{BIN DIRECTORIES REQUIRE 1}]
            cpp.dynamicLibraries: ["{LIB NAMES REQUIRE 1}"]
            cpp.defines: []
            cpp.cppFlags: []
            cpp.cFlags: []
            cpp.linkerFlags: []
        }
    }
    // lib root path: {ROOT PATH REQUIRE 1}

    Product {
        name: "REQUIRE2"
        Export {
            Depends { name: "cpp" }
            cpp.includePaths: [{INCLUDE DIRECTORIES REQUIRE 2}]
            cpp.libraryPaths: [{LIB DIRECTORIES REQUIRE 2}]
            cpp.systemIncludePaths: [{BIN DIRECTORIES REQUIRE 2}]
            cpp.dynamicLibraries: ["{LIB NAMES REQUIRE 2}"]
            cpp.defines: []
            cpp.cppFlags: []
            cpp.cFlags: []
            cpp.linkerFlags: []
        }
    }
    // lib root path: {ROOT PATH REQUIRE 2}
}
```

13.4.11 qmake

This is the reference page for `qmake` generator. Go to *Integrations/Qmake* if you want to learn how to integrate your project or recipes with qmake.

Generates a file named `conanbuildinfo.pri` that can be used for your qbs builds. The file contains:

- N groups of variables, one group per require, declaring the same individual values: include_paths, libs, bin dirs, libraries, defines etc.
- One group of global variables with the aggregated values for all requirements.

Package declared vars

For each requirement `conanbuildinfo.pri` file declares the following variables. `XXX` is the name of the require in uppercase. e.k "ZLIB" for `zlib/1.2.8@lasote/stable` requirement:

NAME	VALUE
CONAN_XXX_ROOT	Abs path to root package folder.
CONAN_INCLUDEPATH_XXX	Header's folders
CONAN_LIB_DIRS_XXX	Library folders (default {CONAN_XXX_ROOT}/lib)
CONAN_BINDIRS_XXX	Binary folders (default {CONAN_XXX_ROOT}/bin)
CONAN_LIBS_XXX	Library names to link
CONAN_DEFINES_XXX	Library defines
CONAN_COMPILE_DEFINITIONS_XXX	Compile definitions
CONAN_QMAKE_CXXFLAGS_XXX	CXX flags
CONAN_QMAKE_LFLAGS_XXX	Shared link flags
CONAN_QMAKE_CFLAGS_XXX	C flags

Global declared vars

Conan also declares some global variables with the aggregated values of all our requirements. The values are ordered in the right order according to the dependency tree.

NAME	VALUE
CONAN_INCLUDEPATH	Aggregated header's folders
CONAN_LIB_DIRS	Aggregated library folders
CONAN_BINDIRS	Aggregated binary folders
CONAN_LIBS	Aggregated library names to link
CONAN_DEFINES	Aggregated library defines
CONAN_COMPILE_DEFINITIONS	Aggregated compile definitions
CONAN_QMAKE_CXXFLAGS	Aggregated CXX flags
CONAN_QMAKE_LFLAGS	Aggregated Shared link flags
CONAN_QMAKE_CFLAGS	Aggregated C flags

Methods available in conanbuildinfo.pri

NAME	DESCRIPTION
conan_basic_setup()	Setup all the qmake vars according to our settings with the global approach

13.4.12 scons

Conan provides *integration with SCons* with this generator.

The generated `SConscript_conan` will generate several dictionaries, like:

```
"conan" : {
    "CPPPATH"     : ['/path/to/include'],
    "LIBPATH"     : ['/path/to/lib'],
    "BINPATH"     : ['/path/to/bin'],
    "LIBS"        : ['hello'],
    "CPPDEFINES"  : [],
    "CXXFLAGS"    : [],
    "CCFLAGS"     : [],
    "SHLINKFLAGS" : [],
```

(continues on next page)

(continued from previous page)

```
    "LINKFLAGS"    : [],
},

"Hello" : {
    "CPPPATH"      : ['/path/to/include'],
    "LIBPATH"      : ['/path/to/lib'],
    "BINPATH"      : ['/path/to/bin'],
    "LIBS"         : ['hello'],
    "CPPDEFINES"   : [],
    "CXXFLAGS"     : [],
    "CCFLAGS"      : [],
    "SHLINKFLAGS"  : [],
    "LINKFLAGS"    : [],
},
```

The `conan` dictionary will contain the aggregated values for all dependencies, while the individual `"Hello"` dictionaries, one per package, will contain just the values for that specific dependency.

These dictionaries can be directly loaded into the environment like:

```
conan = SConscript('{}/SConscript_conan'.format(build_path_relative_to_sconstruct))
env.MergeFlags(conan['conan'])
```

13.4.13 pkg_config

Generates N files named `{dep_name}.pc`, containing a valid pkg-config file syntax. The `prefix` variable is automatically adjusted to the `package_folder`.

Go to *Integrations/pkg-config and pc files/Use the pkg_config generator* if you want to learn how to use this generator.

13.4.14 virtualenv

This is the reference page for `virtualenv` generator. Go to *Mastering/Virtual Environments* if you want to learn how to use conan virtual environments.

Created files

- activate.{sh|bat|ps1}
- deactivate.{sh|bat|ps1}

Usage

Linux/OSX:

```
> source activate.sh
```

Windows:

```
> activate.bat
```

Variables declared

ENVIRONMENT VAR	VALUE
PS1	New shell prompt value corresponding to the current directory name
OLD_PS1	Old PS1 value, to recover it in deactivation
XXXX	Any variable declared in the `self.env_info` object of the requirements.

13.4.15 virtualbuildenv

This is the reference page for `virtualbuildenv` generator. Go to *Mastering/Virtual Environments* if you want to learn how to use conan virtual environments.

Created files

- activate_build.{sh|bat}
- deactivate_build.{sh|bat}

Usage

Linux/OSX:

```
> source activate_build.sh
```

Windows:

```
> activate_build.bat
```

Variables declared

ENVIRONMENT VAR	DESCRIPTION
LIBS	Library names to link
LDFLAGS	Link flags, (-L, -m64, -m32)
CFLAGS	Options for the C compiler (-g, -s, -m64, -m32, -fPIC)
CXXFLAGS	Options for the C++ compiler (-g, -s, -stdlib, -m64, -m32, -fPIC)
CPPFLAGS	Preprocessor definitions (-D, -I)
LIB	Library paths separated with ";" (Visual Studio)
CL	"/I" flags with include directories (Visual Studio)

13.4.16 virtualrunenv

This is the reference page for `virtualrunenv` generator. Go to *Mastering/Virtual Environments* if you want to learn how to use conan virtual environments.

Created files

- activate_run.{sh|bat}
- deactivate_run.{sh|bat}

Usage

Linux/OSX:

```
> source activate_run.sh
```

Windows:

```
> activate_run.bat
```

Variables declared

ENVIRONMENT VAR	DESCRIPTION
PATH	With every `bin` folder of your requirements.
LD_LIBRARY_PATH	`lib` folders of your requirements.
DYLD_LIBRARY_PATH	`lib` folders of your requirements.

13.4.17 youcompleteme

Go to *Integrations/YouCompleteMe* to see the details of the YouCompleteMe generator.

13.4.18 txt

This is the reference page for `txt` generator. Go to *Integrations/Custom integrations / Use the text generator* to know how to use it.

File format

The generated `conanbuildinfo.txt` file is a generic config file with `[sections]` and values.

Package declared vars

For each requirement `conanbuildinfo.txt` file declares the following sections. `XXX` is the name of the require in lowercase. e.k "zlib" for `zlib/1.2.8@lasote/stable` requirement:

SECTION	DESCRIPTION
[include_dirs_XXX]	List with the include paths of the requirement
[libdirs_XXX]	List with library paths of the requirement
[bindirs_XXX]	List with binary directories of the requirement
[resdirs_XXX]	List with the resource directories of the requirement
[builddirs_XXX]	List with the build directories of the requirement
[libs_XXX]	List with library names of the requirement
[defines_XXX]	List with the defines of the requirement
[cflags_XXX]	List with C compilation flags
[sharedlinkflags_XXX]	List with shared libraries link flags
[exelinkflags_XXX]	List with executable link flags
[cppflags_XXX]	List with C++ compilation flags
[rootpath_XXX]	Root path of the package

Global declared vars

Conan also declares some global variables with the aggregated values of all our requirements. The values are ordered in the right order according to the dependency tree.

SECTION	DESCRIPTION
[include_dirs]	List with the aggregated include paths of the requirements
[libdirs]	List with aggregated library paths of the requirements
[bindirs]	List with aggregated binary directories of the requirements
[resdirs]	List with the aggregated resource directories of the requirements
[builddirs]	List with the aggregated build directories of the requirements
[libs]	List with aggregated library names of the requirements
[defines]	List with the aggregated defines of the requirements
[cflags]	List with aggregated C compilation flags
[sharedlinkflags]	List with aggregated shared libraries link flags
[exelinkflags]	List with aggregated executable link flags
[cppflags]	List with aggregated C++ compilation flags

13.4.19 json

A file named *conanbuildinfo.json* will be generated. It will contain the information about every dependency:

```
{
  "deps_env_info": {
    "MY_ENV_VAR": "foo"
  },
  "deps_user_info": {
    "Hello": {
      "my_var": "my_value"
    }
  },
  "dependencies":
  [
    {
      "name": "fmt",
      "version": "4.1.0",
      "include_paths": [
        "/path/to/.conan/data/fmt/4.1.0/<user>/<channel>/package/<id>/include"
```

(continues on next page)

(continued from previous page)

```
      ],
      "lib_paths": [
        "/path/to/.conan/data/fmt/4.1.0/<user>/<channel>/package/<id>/lib"
      ],
      "libs": [
        "fmt"
      ],
      "...": "...",
    },
    {
      "name": "Poco",
      "version": "1.7.8p3",
      "...": "..."
    }
  ],
}
```

The generated `conanbuildinfo.json` file is a json file with the following keys:

dependencies

The dependencies is a list, with each item belonging to one dependency, and each one with the following keys: - name - version - description - rootpath - sysroot - include_paths, lib_paths, bin_paths, build_paths, res_paths - libs - defines, cflags, cppflags, sharedlinkflags, exelinkflags

Plese note it is an ordered list, not a map, and dependency order is relevant. Upstream dependencies, i.e. the ones that do not depend on other packages, will be first, and their direct dependencies after them, and so on.

deps_env_info

The environment variables defined by upstream dependencies

deps_user_info

The user variables defined by upstream dependencies

13.5 Profiles

Profiles allows users to set a complete configurateion set for **settings**, **options**, **environment variables**, and **build requirements** in a file. They have this structure:

```
[settings]
setting=value

[options]
MyLib:shared=True

[env]
env_var=value

[build_requires]
```

(continues on next page)

(continued from previous page)

```
Tool1/0.1@user/channel
Tool2/0.1@user/channel, Tool3/0.1@user/channel
*: Tool4/0.1@user/channel
```

Profile files can be used with `-pr`/`--profile` option in **`conan install`** and **`conan create`** commands.

```
$ conan create demo/testing -pr=myprofile
```

Profiles can be located in different folders, for example, the default `<userhome>/.conan/profiles`, and be referenced by absolute or relative path:

```
$ conan install --profile /abs/path/to/profile  # abs path
$ conan install --profile ./relpath/to/profile  # resolved to current dir
$ conan install --profile profile  # resolved to user/.conan/profiles/profile
```

Listing existing profiles can be done like this:

```
$ conan profile list
default
myprofile1
myprofile2
...
```

You can also show profile's content:

```
$ conan profile show myprofile1
Configuration for profile myprofile1:

[settings]
    os=Windows
    arch=x86_64
    compiler=Visual Studio
    compiler.version=15
    build_type=Release
[options]
[build_requires]
[env]
```

Use `$PROFILE_DIR` in your profile and it will be replaced with the absolute path to the profile file. It is useful to declare relative folders:

```
[env]
PYTHONPATH=$PROFILE_DIR/my_python_tools
```

13.5.1 Package settings and env vars

Profiles also support **package settings** and **package environment variables** definition, so you can override some settings or environment variables for some specific package:

Listing 4: *.conan/profiles/zlib_with_clang*

```
 [settings]
 zlib:compiler=clang
 zlib:compiler.version=3.5
```

(continues on next page)

(continued from previous page)

```
zlib:compiler.libcxx=libstdc++11
compiler=gcc
compiler.version=4.9
compiler.libcxx=libstdc++11

[env]
zlib:CC=/usr/bin/clang
zlib:CXX=/usr/bin/clang++
```

Your build tool will locate **clang** compiler only for the **zlib** package and **gcc** (default one) for the rest of your dependency tree.

Note: If you want to override existing system environment variables, you should use the `key=value` syntax. If you need to pre-pend to the system environment variables you should use the syntax `key=[value]` or `key=[value1, value2, ...]`. A typical example is the `PATH` environment variable, when you want to add paths to the existing system PATH, not override it, you would use:

```
[env]
PATH=[/some/path/to/my/tool]
```

13.5.2 Profile includes

You can include other profiles using the `include()` statement. The path can be relative to the current profile, absolute, or a profile name from the default profile location in the local cache.

The `include()` statement has to be at the top of the profile file:

Listing 5: *gcc_49*

```
[settings]
compiler=gcc
compiler.version=4.9
compiler.libcxx=libstdc++11
```

Listing 6: *myprofile*

```
include(gcc_49)

[settings]
zlib:compiler=clang
zlib:compiler.version=3.5
zlib:compiler.libcxx=libstdc++11

[env]
zlib:CC=/usr/bin/clang
zlib:CXX=/usr/bin/clang++
```

13.5.3 Variable declaration

In a profile you can declare variables that will be replaced automatically by conan before the profile is applied. The variables have to be declared at the top of the file, after the `include()` statements.

Listing 7: *myprofile*

```
include(gcc_49)
CLANG=/usr/bin/clang

[settings]
zlib:compiler=clang
zlib:compiler.version=3.5
zlib:compiler.libcxx=libstdc++11

[env]
zlib:CC=$CLANG/clang
zlib:CXX=$CLANG/clang++
```

The variables will be inherited too, so you can declare variables in a profile and then include the profile in a different one, all the variables will be available:

Listing 8: *gcc_49*

```
GCC_PATH=/my/custom/toolchain/path/

[settings]
compiler=gcc
compiler.version=4.9
compiler.libcxx=libstdc++11
```

Listing 9: *myprofile*

```
include(gcc_49)

[settings]
zlib:compiler=clang
zlib:compiler.version=3.5
zlib:compiler.libcxx=libstdc++11

[env]
zlib:CC=$GCC_PATH/gcc
zlib:CXX=$GCC_PATH/g++
```

13.5.4 Examples

If you are working with Linux and you usually work with `gcc` compiler, but you have installed `clang` compiler and want to install some package for `clang` compiler, you could do:

- Create a `.conan/profiles/clang` file:

```
[settings]
compiler=clang
compiler.version=3.5
compiler.libcxx=libstdc++11

[env]
CC=/usr/bin/clang
CXX=/usr/bin/clang++
```

- Execute conan install command passing the `--profile` or `-pr` parameter:

```
conan install --profile clang
```

Without profiles you would have needed to set the CC and CXX variables in the environment to point to your clang compiler and use `-s` parameters to specify the settings:

```
export CC=/usr/bin/clang
export CXX=/usr/bin/clang++
conan install -s compiler=clang -s compiler.version=3.5 -s compiler.libcxx=libstdc++11
```

A profile can also be used in **`conan create`** and **`conan info`**:

```
$ conan create demo/testing --profile clang
```

See also:

- Check the section *Build requirements* to read more about its ussage in a profile.
- Check *conan profile* for full reference.
- Check *profiles/default* for full reference.
- Related section: *Cross building*.

13.6 Build helpers

There are several helpers that can assist to automate the build() method for popular build systems

Contents:

13.6.1 CMake

The *CMake* class helps us to invoke *cmake* command with the generator, flags and definitions, reflecting the specified Conan settings.

There are two ways to invoke your cmake tools:

- Using the helper attributes `cmake.command_line` and `cmake.build_config`:

```
from conans import ConanFile, CMake

 class ExampleConan(ConanFile):
     ...

     def build(self):
         cmake = CMake(self)
         self.run('cmake "%s" %s' % (self.source_folder, cmake.command_line))
         self.run('cmake --build . %s' % cmake.build_config)
         self.run('cmake --build . --target install')
```

- Using the helper methods:

```
from conans import ConanFile, CMake

class ExampleConan(ConanFile):
    ...
```

(continues on next page)

(continued from previous page)

```
    def build(self):
        cmake = CMake(self)
        # same as cmake.configure(source_folder=self.source_folder, build_folder=self.
↪build_folder)
        cmake.configure()
        cmake.build()
        cmake.test() # Build the "RUN_TESTS" or "test" target
        # Build the "install" target, defining CMAKE_INSTALL_PREFIX to self.package_
↪folder
        cmake.install()
```

Constructor

```
class CMake(object):

    def __init__(self, conanfile, generator=None, cmake_system_name=True,
                 parallel=True, build_type=None, toolset=None)
```

Parameters:

- **conanfile** (Required): Conanfile object. Usually `self` in a conanfile.py
- **generator** (Optional, Defaulted to `None`): Specify a custom generator instead of autodetect it. e.j: "MinGW Makefiles"
- **cmake_system_name** (Optional, Defaulted to `True`): Specify a custom value for `CMAKE_SYSTEM_NAME` instead of autodetect it.
- **parallel** (Optional, Defaulted to `True`): If `True`, will append the *-jN* attribute for parallel building being N the *cpu_count()*.
- **build_type** (Optional, Defaulted to `None`): Force the build type to be declared in `CMAKE_BUILD_TYPE`. If you set this parameter the build type not will be taken from the settings.
- **toolset** (Optional, Defaulted to `None`): Specify a toolset for Visual Studio.

Attributes

verbose

Defaulted to: `False`

Set it to `True` or `False` to automatically set the definition `CMAKE_VERBOSE_MAKEFILE`.

```
from conans import ConanFile, CMake

class ExampleConan(ConanFile):
    ...

    def build(self):
        cmake = CMake(self)
        cmake.verbose = True
        cmake.configure()
        cmake.build()
```

command_line (read only)

Generator, conan definitions and flags that reflects the specified Conan settings.

```
-G "Unix Makefiles" -DCMAKE_BUILD_TYPE=Release ... -DCONAN_C_FLAGS=-m64 -Wno-dev
```

build_config (read only)

Value for `--config` option for Multi-configuration IDEs.

```
--config Release
```

definitions

The CMake helper will automatically append some definitions based on your settings:

Variable	Description
CMAKE_BUILD_TYPE	Debug or Release (from self.settings.build_type)
CMAKE_OSX_ARCHITECTURES	"i386" if architecture is x86 in an OSX system
BUILD_SHARED_LIBS	Only If your conanfile has a "shared" option
CONAN_COMPILER	Conan internal variable to check compiler
CMAKE_SYSTEM_NAME	If detected cross building it's set to self.settings.os
CMAKE_SYSTEM_VERSION	If detected cross building it's set to the self.settings.os_version
CMAKE_ANDROID_ARCH_ABI	If detected cross building to Android
CONAN_LIBCXX	from self.settings.compiler.libcxx
CONAN_CMAKE_SYSTEM_PROCESSOR	Definition only set if same environment variable is declared by user
CONAN_CMAKE_FIND_ROOT_PATH	Definition only set if same environment variable is declared by user
CONAN_CMAKE_FIND_ROOT_PATH_MODE_PROGRAM	Definition only set if same environment variable is declared by user
CONAN_CMAKE_FIND_ROOT_PATH_MODE_LIBRARY	Definition only set if same environment variable is declared by user
CONAN_CMAKE_FIND_ROOT_PATH_MODE_INCLUDE	Definition only set if same environment variable is declared by user
CONAN_CMAKE_POSITION_INDEPENDENT_CODE	When `fPIC` option is present and True or when `fPIC` is present and False but and option `shared` is present and True
CONAN_SHARED_LINKER_FLAGS	-m32 and -m64 based on your architecture
CONAN_C_FLAGS	-m32 and -m64 based on your architecture and /MP for MSVS
CONAN_C_FLAGS	-m32 and -m64 based on your architecture and /MP for MSVS
CONAN_LINK_RUNTIME	Runtime from self.settings.compiler.runtime for MSVS
CONAN_CMAKE_CXX_STANDARD	From setting `cppstd`
CONAN_CMAKE_CXX_EXTENSIONS	From setting `cppstd`, when GNU extensions are enabled
CONAN_STD_CXX_FLAG	From setting `cppstd`. Flag for compiler directly (for CMake < 3.1)

But you can change the automatic definitions after the `CMake()` object creation using the `definitions` property:

```
from conans import ConanFile, CMake

class ExampleConan(ConanFile):
    ...

    def build(self):
        cmake = CMake(self)
        cmake.definitions["CMAKE_SYSTEM_NAME"] = "Generic"
```

(continues on next page)

(continued from previous page)

```
        cmake.configure()
        cmake.build()
        cmake.install() # Build --target=install
```

Methods

configure()

```
def configure(self, args=None, defs=None, source_folder=None, build_folder=None,
              cache_build_folder=None)
```

Configures *CMake* project with the given parameters.

Parameters:

- **args** (Optional, Defaulted to `None`): A list of additional arguments to be passed to the `cmake` command. Each argument will be escaped according to the current shell. No extra arguments will be added if `args=None`
- **definitions** (Optional, Defaulted to `None`): A dict that will be converted to a list of CMake command line variable definitions of the form `-DKEY=VALUE`. Each value will be escaped according to the current shell and can be either `str`, `bool` or of numeric type
- **source_folder**: CMake's source directory where `CMakeLists.txt` is located. The default value is the `self.source_folder`. Relative paths are allowed and will be relative to `self.source_folder`.
- **build_folder**: CMake's output directory. The default value is the `self.build_folder` if `None` is specified. The `CMake` object will store `build_folder` internally for subsequent calls to `build()`.
- **cache_build_folder** (Optional, Defaulted to `None`): Use the given subfolder as build folder when building the package in the local cache. This argument doesn't have effect when the package is being built in user folder with **`conan build`** but overrides **build_folder** when working in the local cache. See *self.in_local_cache*.

build()

```
def build(self, args=None, build_dir=None, target=None)
```

Builds *CMake* project with the given parameters.

Parameters:

- **args** (Optional, Defaulted to `None`): A list of additional arguments to be passed to the `cmake` command. Each argument will be escaped according to the current shell. No extra arguments will be added if `args=None`
- **build_dir** (Optional, Defaulted to `None`): CMake's output directory. If `None` is specified the `build_dir` from `configure()` will be used.
- **target** (Optional, Defaulted to `None`): Specifies the target to execute. The default *all* target will be built if `None` is specified. `"install"` can be used to relocate files to aid packaging.

test()

```
def test(args=None, build_dir=None, target=None)
```

Build *CMake* test target (could be RUN_TESTS in multi-config projects or `test` in single-config projects), which usually means building and running unit tests

Parameters:

- **args** (Optional, Defaulted to `None`): A list of additional arguments to be passed to the `cmake` command. Each argument will be escaped according to the current shell. No extra arguments will be added if `args=None`.
- **build_dir** (Optional, Defaulted to `None`): CMake's output directory. If `None` is specified the `build_folder` from `configure()` will be used.
- **target** (Optional, default to `None`). Alternative target name for running the tests. If not defined RUN_TESTS or `test` will be used

install()

```
def install(args=None, build_dir=None)
```

Installs *CMake* project with the given parameters.

Parameters:

- **args** (Optional, Defaulted to `None`): A list of additional arguments to be passed to the `cmake` command. Each argument will be escaped according to the current shell. No extra arguments will be added if `args=None`.
- **build_dir** (Optional, Defaulted to `None`): CMake's output directory. If `None` is specified the `build_folder` from `configure()` will be used.

patch_config_paths() [EXPERIMENTAL]

```
def patch_config_paths()
```

This method changes references to the absolute path of the installed package in exported CMake config files to the appropriate Conan variable. This makes most CMake config files portable.

For example, if a package foo installs a file called *fooConfig.cmake* to be used by cmake's `find_package()` method, normally this file will contain absolute paths to the installed package folder, for example it will contain a line such as:

```
SET(Foo_INSTALL_DIR /home/developer/.conan/data/Foo/1.0.0/...)
```

This will cause cmake's `find_package()` method to fail when someone else installs the package via Conan. This function will replace such paths to:

```
SET(Foo_INSTALL_DIR ${CONAN_FOO_ROOT})
```

Which is a variable that is set by *conanbuildinfo.cmake*, so that `find_package()` now correctly works on this Conan package.

If the `install()` method of the CMake object in the conanfile is used, this function should be called **after** that invocation. For example:

```
def build(self):
    cmake = CMake(self)
    cmake.configure()
    cmake.build()
    cmake.install()
    cmake.patch_config_paths()
```

Environment variables

There are some environment variables that will also affect the `CMake()` helper class. Check them in the *CMAKE RELATED VARIABLES* section.

13.6.2 AutoToolsBuildEnvironment (configure/make)

If you are using **configure/make** you can use **AutoToolsBuildEnvironment** helper. This helper sets `LIBS`, `LDFLAGS`, `CFLAGS`, `CXXFLAGS` and `CPPFLAGS` environment variables based on your requirements.

```
from conans import ConanFile, AutoToolsBuildEnvironment

class ExampleConan(ConanFile):
   settings = "os", "compiler", "build_type", "arch"
   requires = "Poco/1.9.0@pocoproject/stable"
   default_options = "Poco:shared=True", "OpenSSL:shared=True"

   def imports(self):
      self.copy("*.dll", dst="bin", src="bin")
      self.copy("*.dylib*", dst="bin", src="lib")

   def build(self):
      autotools = AutoToolsBuildEnvironment(self)
      autotools.configure()
      autotools.make()
```

It also works using the *environment_append* context manager applied to your **configure and make** commands, calling *configure* and *make* manually:

```
from conans import ConanFile, AutoToolsBuildEnvironment

class ExampleConan(ConanFile):
    ...

    def build(self):
        env_build = AutoToolsBuildEnvironment(self)
        with tools.environment_append(env_build.vars):
            self.run("./configure")
            self.run("make")
```

You can change some variables like `fpic`, `libs`, `include_paths` and `defines` before accessing the `vars` to override an automatic value or add new values:

```
from conans import ConanFile, AutoToolsBuildEnvironment
```

(continues on next page)

(continued from previous page)

```
class ExampleConan(ConanFile):
    ...

    def build(self):
        env_build = AutoToolsBuildEnvironment(self)
        env_build.fpic = True
        env_build.libs.append("pthread")
        env_build.defines.append("NEW_DEFINE=23")
        env_build.configure()
        env_build.make()
```

You can use it also with `MSYS2/MinGW` subsystems installed by setting the *win_bash* parameter in the constructor. It will run the the `configure` and `make` commands inside a `bash` that has to be in the path or declared in `CONAN_BASH_PATH`:

```
from conans import ConanFile, AutoToolsBuildEnvironment
import platform

class ExampleConan(ConanFile):
   settings = "os", "compiler", "build_type", "arch"

   def imports(self):
     self.copy("*.dll", dst="bin", src="bin")
     self.copy("*.dylib*", dst="bin", src="lib")

   def build(self):
      in_win = platform.system() == "Windows"
      env_build = AutoToolsBuildEnvironment(self, win_bash=in_win)
      env_build.configure()
      env_build.make()
```

Constructor

```
class AutoToolsBuildEnvironment(object):

    def __init__(self, conanfile, win_bash=False)
```

Parameters:

- **conanfile** (Required): Conanfile object. Usually `self` in a conanfile.py
- **win_bash**: (Optional, Defaulted to `False`): When True, it will run the configure/make commands inside a bash.

Attributes

You can adjust the automatically filled values modifying the attributes like this:

```
from conans import ConanFile, AutoToolsBuildEnvironment

class ExampleConan(ConanFile):
    ...

    def build(self):
```

(continues on next page)

(continued from previous page)

```
        autotools = AutoToolsBuildEnvironment(self)
        autotools.fpic = True
        autotools.libs.append("pthread")
        autotools.defines.append("NEW_DEFINE=23")
        autotools.configure()
        autotools.make()
```

fpic

Defaulted to: `None`

Set it to `True` if you want to append the `-fPIC` flag.

libs

List with library names of the requirements (`-l` in `LIBS`).

include_paths

List with the include paths of the requires (-I in CPPFLAGS).

library_paths

List with library paths of the requirements (-L in LDFLAGS).

defines

List with variables that will be defined with `-D` in `CPPFLAGS`.

flags

List with compilation flags (`CFLAGS` and `CXXFLAGS`).

cxx_flags

List with only C++ compilation flags (`CXXFLAGS`).

link_flags

List with linker flags

Properties

vars

Environment variables `CPPFLAGS`, `CXXFLAGS`, `CFLAGS`, `LDFLAGS`, `LIBS` generated by the build helper to use them in the configure, make and install steps. This variables are generated dynamically with the values of the attributes and can also be modified to be used in the following configure, make or install steps:

```
def build():
    auotools = AutoToolsBuildEnvironment()
    autotools.fpic = True
    env_build_vars = autotools.vars
    env_build_vars['RCFLAGS'] = '-O COFF'
    autotools.configure(vars=env_build_vars)
    autotools.make(vars=env_build_vars)
    autotools.install(vars=env_build_vars)
```

vars_dict

Same behavior as `vars` but this property returns each variable `CPPFLAGS`, `CXXFLAGS`, `CFLAGS`, `LDFLAGS`, `LIBS` as dictionaries.

Methods

configure()

```
def configure(self, configure_dir=None, args=None, build=None, host=None, target=None,
↪ pkg_config_paths=None, vars=None)
```

Configures *Autotools* project with the given parameters.

Important: This method sets by default the `--prefix` argument to `self.package_folder` whenever `--prefix` is not provided in the `args` parameter during the configure step.

Parameters:

- **configure_dir** (Optional, Defaulted to `None`): Directory where the `configure` script is. If `None`, it will use the current directory.
- **args** (Optional, Defaulted to `None`): A list of additional arguments to be passed to the `configure` script. Each argument will be escaped according to the current shell. No extra arguments will be added if `args=None`.
- **build** (Optional, Defaulted to `None`): To specify a value for the parameter `--build`. If `None` it will try to detect the value if cross-building is detected according to the settings. If `False`, it will not use this argument at all.
- **host** (Optional, Defaulted to `None`): To specify a value for the parameter `--host`. If `None` it will try to detect the value if cross-building is detected according to the settings. If `False`, it will not use this argument at all.

- **target** (Optional, Defaulted to `None`): To specify a value for the parameter `--target`. If `None` it will try to detect the value if cross-building is detected according to the settings. If `False`, it will not use this argument at all.
- **pkg_config_paths** (Optional, Defaulted to `None`): To specify folders (in a list) where to find `*.pc` files (by using the env var `PKG_CONFIG_PATH`). If `None` is specified but the conanfile is using the `pkg_config` generator, the `self.build_folder` will be added to the `PKG_CONFIG_PATH` in order to locate the pc files of the requirements of the conanfile.
- **vars** (Optional, Defaulted to `None`): Overrides custom environment variables in the configure step.

make()

```
def make(self, args="", make_program=None, target=None, vars=None)
```

Builds *Autotools* project with the given parameters.

Parameters:

- **args** (Optional, Defaulted to `""`): A list of additional arguments to be passed to the `make` command. Each argument will be escaped accordingly to the current shell. No extra arguments will be added if `args=""`.
- **make_program** (Optional, Defaulted to `None`): Allows to specify a different `make` executable, e.j: `mingw32-make`. The environment variable *CONAN_MAKE_PROGRAM* can be used too.
- **target** (Optional, Defaulted to `None`): Choose which target to build. This allows building of e.g. docs, shared libraries or install for some AutoTools projects.
- **vars** (Optional, Defaulted to `None`): Overrides custom environment variables in the make step.

install()

```
def install(self, args="", make_program=None, vars=None)
```

Performs the install step of autotools calling `make(target="install")`.

Paramenters:

- **args** (Optional, Defaulted to `""`): A list of additional arguments to be passed to the `make` command. Each argument will be escaped accordingly to the current shell. No extra arguments will be added if `args=""`.
- **make_program** (Optional, Defaulted to `None`): Allows to specify a different `make` executable, e.j: `mingw32-make`. The environment variable *CONAN_MAKE_PROGRAM* can be used too.
- **vars** (Optional, Defaulted to `None`): Overrides custom environment variables in the install step.

Environment variables

The following environment variables will also affect the *AutoToolsBuildEnvironment* helper class.

NAME	DESCRIPTION
LIBS	Library names to link
LDFLAGS	Link flags, (-L, -m64, -m32)
CFLAGS	Options for the C compiler (-g, -s, -m64, -m32, -fPIC)
CXXFLAGS	Options for the C++ compiler (-g, -s, -stdlib, -m64, -m32, -fPIC, -std)
CPPFLAGS	Preprocessor definitions (-D, -I)

See also:

- *Reference/Tools/environment_append*

13.6.3 MSBuild

Calls Visual Studio `msbuild` command to build a `sln` project:

```
from conans import ConanFile, MSBuild

class ExampleConan(ConanFile):
    ...

    def build(self):
        msbuild = MSBuild(self)
        msbuild.build("MyProject.sln")
```

Internally the `MSBuild` build helper uses:

- *VisualStudioBuildEnvironment* to adjust the `LIB` and `CL` environment variables with all the information from the requirements: include directories, library names, flags etc.
- *tools.msvc_build_command* to call msbuild.

You can adjust all the information from the requirements accessing to the `build_env` that it is a *VisualStudioBuildEnvironment* object:

```
from conans import ConanFile, MSBuild

class ExampleConan(ConanFile):
    ...

    def build(self):
        msbuild = MSBuild(self)
        msbuild.build_env.include_paths.append("mycustom/directory/to/headers")
        msbuild.build_env.lib_paths.append("mycustom/directory/to/libs")
        msbuild.build_env.link_flags = []

        msbuild.build("MyProject.sln")
```

Constructor

```
class MSBuild(object):

    def __init__(self, conanfile)
```

Parameters:

- **conanfile** (Required): ConanFile object. Usually `self` in a `conanfile.py`.

Methods

build()

```
def build(self, project_file, targets=None, upgrade_project=True, build_type=None,␣
↪arch=None,
          parallel=True, force_vcvars=False, toolset=None, platforms=None, use_
↪env=True)
```

Builds Visual Studio project with the given parameters. It will call `tools.msvc_build_command()`.

Parameters:

- **project_file** (Required): Path to the `sln` file.
- **targets** (Optional, Defaulted to `None`): List of targets to build.
- **upgrade_project** (Optional, Defaulted to `True`): Will call `devenv` to upgrade the solution to your current Visual Studio.
- **build_type** (Optional, Defaulted to `None`): Optional. Defaulted to None, will use the `settings.build_type`
- **arch** (Optional, Defaulted to `None`): Optional. Defaulted to None, will use `settings.arch`
- **force_vcvars** (Optional, Defaulted to `False`): Will ignore if the environment is already set for a different Visual Studio version.
- **parallel** (Optional, Defaulted to `True`): Will use the configured number of cores in the *conan.conf* file (`cpu_count`).
- **toolset** (Optional, Defaulted to `None`): Specify a toolset. Will append a `/p:PlatformToolset` option.
- **platforms** (Optional, Defaulted to `None`): Dictionary with the mapping of archs/platforms from Conan naming to another one. It is useful for Visual Studio solutions that have a different naming in architectures. Example: `platforms={"x86":"Win32"}` (Visual solution uses "Win32" instead of "x86"). This dictionary will update the default one:

```
msvc_arch = {'x86': 'x86',
             'x86_64': 'x64',
             'armv7': 'ARM',
             'armv8': 'ARM64'}
```

- **use_env** (Optional, Defaulted to `True`: Applies the argument `/p:UseEnv=true` to the `msbuild()` call.

get_command()

Returns a string command calling `msbuild`

```
def get_command(self, project_file, props_file_path=None, targets=None, upgrade_
↪project=True, build_type=None,
                arch=None, parallel=True, toolset=None, platforms=None, use_
↪env=False):
```

Parameters:

- **project_file** (Optional, defaulted to None): Path to a properties file to include in the project.
- Same other parameters than **build()**

13.6.4 VisualStudioBuildEnvironment

Prepares the needed environment variables to invoke the Visual Studio compiler. Use it together with *vcvars_command tool*

```
from conans import ConanFile, VisualStudioBuildEnvironment

class ExampleConan(ConanFile):

    ...

    def build(self):
        if self.settings.compiler == "Visual Studio":
            env_build = VisualStudioBuildEnvironment(self)
            with tools.environment_append(env_build.vars):
                vcvars = tools.vcvars_command(self.settings)
                self.run('%s && cl /c /EHsc hello.cpp' % vcvars)
                self.run('%s && lib hello.obj -OUT:hello.lib' % vcvars
```

Set environment variables:

NAME	DESCRIPTION
LIB	Library paths separated with ";"
CL	"/I" flags with include directories, Runtime (/MT, /MD...), Definitions (/DXXX), and any other C and CXX flags.

Attributes

PROPERTY	DESCRIPTION
.include_paths	List with directories of include paths
.lib_paths	List with directories of libraries
.defines	List with definitions (from requirements cpp_info.defines)
.runtime	List with directories (from settings.compiler.runtime)
.flags	List with flag (from requirements cpp_info.cflags
.cxx_flags	List with cxx flags (from requirements cpp_info.cppflags
.link_flags	List with linker flags (from requirements cpp_info.sharedlinkflags and cpp_info.exelinkflags

You can adjust the automatically filled values modifying the attributes above:

```
   def build(self):
      if self.settings.compiler == "Visual Studio":
         env_build = VisualStudioBuildEnvironment(self)
         env_build.include_paths.append("mycustom/directory/to/headers")
         env_build.lib_paths.append("mycustom/directory/to/libs")
         env_build.link_flags = []
         with tools.environment_append(env_build.vars):
             vcvars = tools.vcvars_command(self.settings)
             self.run('%s && cl /c /EHsc hello.cpp' % vcvars)
             self.run('%s && lib hello.obj -OUT:hello.lib' % vcvars
```

See also:

- *Reference/Tools/environment_append*

13.6.5 Meson

If you are using **Meson Build** as your build system, you can use the **Meson** build helper. Specially useful with the *pkg_config generator* that will generate the `*.pc` files of our requirements, then `Meson()` build helper will locate them automatically.

```
from conans import ConanFile, tools, Meson
import os

class ConanFileToolsTest(ConanFile):
    generators = "pkg_config"
    requires = "LIB_A/0.1@conan/stable"
    settings = "os", "compiler", "build_type"

    def build(self):
        meson = Meson(self)
        meson.configure()
        meson.build()
```

Constructor

```
class Meson(object):

    def __init__(self, conanfile, backend=None, build_type=None)
```

Parameters:

- **conanfile** (Required): Use `self` inside a `conanfile.py`.
- **backend** (Optional, Defaulted to `None`): Specify a backend to be used, otherwise it will use `"Ninja"`.
- **build_type** (Optional, Defaulted to `None`): Force to use a build type, ignoring the value from the settings.

Methods

configure()

```
def configure(self, args=None, defs=None, source_folder=None, build_folder=None,
              pkg_config_paths=None, cache_build_folder=None)
```

Configures *Meson* project with the given parameters.

Parameters:

- **args** (Optional, Defaulted to `None`): A list of additional arguments to be passed to the `configure` script. Each argument will be escaped according to the current shell. No extra arguments will be added if `args=None`.
- **defs** (Optional, Defaulted to `None`): A list of definitions.
- **source_folder** (Optional, Defaulted to `None`): Meson's source directory where `meson.build` is located. The default value is the `self.source_folder`. Relative paths are allowed and will be relative to `self.source_folder`.
- **build_folder** (Optional, Defaulted to `None`): Meson's output directory. The default value is the `self.build_folder` if `None` is specified. The `Meson` object will store `build_folder` internally for subsequent calls to `build()`.

- **pkg_config_paths** (Optional, Defaulted to `None`): A list containing paths to locate the pkg-config files (*.pc). If `None`, it will be set to `conanfile.build_folder`.
- **cache_build_folder** (Optional, Defaulted to `None`): Subfolder to be used as build folder when building the package in the local cache. This argument doesn't have effect when the package is being built in user folder with **`conan build`** but overrides **build_folder** when working in the local cache. See *self.in_local_cache*.

build()

```
def build(self, args=None, build_dir=None, targets=None)
```

Builds *Meson* project with the given parameters.

Parameters:

- **args** (Optional, Defaulted to `None`): A list of additional arguments to be passed to the `make` command. Each argument will be escaped according to the current shell. No extra arguments will be added if `args=None`.
- **build_dir** (Optional, Defaulted to `None`): Build folder. If `None`, it will be set to `conanfile.build_folder`.
- **targets** (Optional, Defaulted to `None`): A list of targets to be built. No targets will be added if `targets=None`.

Example

A typical usage of the Meson build helper, if you want to be able to both execute **`conan create`** and also build your package for a library locally (in your user folder, not in the local cache), could be:

```
from conans import ConanFile, Meson

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

    def build(self):
        meson = Meson(self)
        meson.configure(source_folder="%s/src" % self.source_folder,
                        build_folder="build")
        meson.build()

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

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

Note the **pkg_config** generator, which generates .pc files, which are understood by Meson to process dependencies informations (no need for a "meson" generator).

The layout is:

```
<folder>
  | - conanfile.py
  | - src
      | - meson.build
      | - hello.cpp
      | - hello.h
```

And the `meson.build` could be as simple as:

```
project('hello', 'cpp', version : '0.1.0',
                 default_options : ['cpp_std=c++11'])

library('hello', ['hello.cpp'])
```

This allows, to create the package with **`conan create`** as well as to build the package locally:

```
$ cd <folder>
$ conan create user/testing
# Now local build
$ mkdir build && cd build
$ conan install ..
$ conan build ..
```

13.6.6 RunEnvironment

The `RunEnvironment` helper prepare `PATH`, `LD_LIBRARY_PATH` and `DYLD_LIBRARY_PATH` environment variables to locate shared libraries and executables of your requirements at runtime.

This helper is specially useful:

- If you are requiring packages with shared libraries and you are running some executable that needs those libraries.
- If you have a requirement with some tool (executable) and you need it in the path.

```
from conans import ConanFile, RunEnvironment

class ExampleConan(ConanFile):
    ...

    def build(self):
        env_build = RunEnvironment(self)
        with tools.environment_append(env_build.vars):
            self.run("....")
            # All the requirements bin folder will be available at PATH
            # All the lib folders will be available in LD_LIBRARY_PATH and DYLD_LIBRARY_
↪PATH
```

Set environment variables:

NAME	DESCRIPTION
PATH	Containing all the requirements `bin` folders.
LD_LIBRARY_PATH	Containing all the requirements `lib` folders. (Linux)
DYLD_LIBRARY_PATH	Containing all the requirements `lib` folders. (OSX)

See also:

- *Reference/Tools/environment_append*

13.7 Tools

Under the tools module there are several functions and utilities that can be used in conan package recipes:

```
from conans import ConanFile
from conans import tools

class ExampleConan(ConanFile):
    ...
```

13.7.1 tools.cpu_count()

```
def tools.cpu_count()
```

Returns the number of CPUs available, for parallel builds. If processor detection is not enabled, it will safely return 1. Can be overwritten with the environment variable `CONAN_CPU_COUNT` and configured in the *conan.conf file*.

13.7.2 tools.vcvars_command()

```
def vcvars_command(settings, arch=None, compiler_version=None, force=False)
```

Returns, for given settings, the command that should be called to load the Visual Studio environment variables for a certain Visual Studio version. It does not execute the command, as that typically have to be done in the same command as the compilation, so the variables are loaded for the same subprocess. It will be typically used in the `build()` method, like this:

```
from conans import tools

def build(self):
    if self.settings.build_os == "Windows":
        vcvars = tools.vcvars_command(self.settings)
        build_command = ...
        self.run("%s && configure %s" % (vcvars, " ".join(args)))
        self.run("%s && %s %s" % (vcvars, build_command, " ".join(build_args)))
```

The `vcvars_command` string will contain something like `call "%vsXX0comntools%../../VC/vcvarsall.bat"` for the corresponding Visual Studio version for the current settings.

This is typically not needed if using `CMake`, as the cmake generator will handle the correct Visual Studio version.

If **arch** or **compiler_version** is specified, it will ignore the settings and return the command to set the Visual Studio environment for these parameters.

Parameters:

- **settings** (Required): Conanfile settings. Use `self.settings`.
- **arch** (Optional, Defaulted to `None`): Will use `settings.arch`.
- **compiler_version** (Optional, Defaulted to `None`): Will use `settings.compiler.version`.
- **force** (Optional, Defaulted to `False`): Will ignore if the environment is already set for a different Visual Studio version.

13.7.3 tools.vcvars_dict()

```
vcvars_dict(settings, arch=None, compiler_version=None, force=False, filter_known_
↪paths=False)
```

Returns a dictionary with the variables set by the **tools.vcvars_command**.

```
from conans import tools

def build(self):
    env_vars = tools.vcvars_dict(self.settings):
    with tools.environment_append(env_vars):
        # Do something
```

Parameters:

- Same as `vcvars_command`.
- **filter_known_paths** (Optional, Defaulted to `False`): When True, the function will only keep the PATH entries that follows some known patterns, filtering all the non-Visual Studio ones. When False, it will keep the PATH will all the system entries.

13.7.4 tools.vcvars()

```
vcvars(settings, arch=None, compiler_version=None, force=False, filter_known_
↪paths=False)
```

Note: This context manager tool has no effect if used in a platform different from Windows.

This is a context manager that allows to append to the environment all the variables set by the **tools.vcvars_dict()**. You can replace **tools.vcvars_command()** and use this context manager to get a cleaner way to activate the Visual Studio environment:

```
from conans import tools

def build(self):
    with tools.vcvars(self.settings):
        do_something()
```

13.7.5 tools.build_sln_command() (DEPRECATED)

Warning: This tool is deprecated and will be removed in Conan 2.0. Use *MSBuild()* build helper instead.

```
def build_sln_command(settings, sln_path, targets=None, upgrade_project=True, build_
↪type=None,
                      arch=None, parallel=True, toolset=None, platforms=None)
```

Returns the command to call *devenv* and *msbuild* to build a Visual Studio project. It's recommended to use it along with `vcvars_command()`, so that the Visual Studio tools will be in path.

```
from conans import tools

def build(self):
    build_command = build_sln_command(self.settings, "myfile.sln", targets=["SDL2_
↪image"])
    command = "%s && %s" % (tools.vcvars_command(self.settings), build_command)
    self.run(command)
```

Parameters:

- **settings** (Required): Conanfile settings. Use "self.settings".
- **sln_path** (Required): Visual Studio project file path.
- **targets** (Optional, Defaulted to `None`): List of targets to build.
- **upgrade_project** (Optional, Defaulted to `True`): If `True`, the project file will be upgraded if the project's VS version is older than current. When *CONAN_SKIP_VS_PROJECTS_UPGRADE* environment variable is set to `True`/`1`, this parameter will be ignored and the project won't be upgraded.
- **build_type** (Optional, Defaulted to `None`): Override the build type defined in the settings (`settings.build_type`).
- **arch** (Optional, Defaulted to `None`): Override the architecture defined in the settings (`settings.arch`).
- **parallel** (Optional, Defaulted to `True`): Enables VS parallel build with `/m:X` argument, where X is defined by CONAN_CPU_COUNT environment variable or by the number of cores in the processor by default.
- **toolset** (Optional, Defaulted to `None`): Specify a toolset. Will append a `/p:PlatformToolset` option.
- **platforms** (Optional, Defaulted to `None`): Dictionary with the mapping of archs/platforms from Conan naming to another one. It is useful for Visual Studio solutions that have a different naming in architectures. Example: `platforms={"x86":"Win32"}` (Visual solution uses "Win32" instead of "x86"). This dictionary will update the default one:

```
msvc_arch = {'x86': 'x86',
             'x86_64': 'x64',
             'armv7': 'ARM',
             'armv8': 'ARM64'}
```

13.7.6 tools.msvc_build_command() (DEPRECATED)

Warning: This tool is deprecated and will be removed in Conan 2.0. Use *MSBuild()*.get_command() instead.

```
def msvc_build_command(settings, sln_path, targets=None, upgrade_project=True, build_
↪type=None,
                       arch=None, parallel=True, force_vcvars=False, toolset=None,␣
↪platforms=None)
```

Returns a string with a joint command consisting in setting the environment variables via `vcvars.bat` with the above `tools.vcvars_command()` function, and building a Visual Studio project with the `tools.build_sln_command()` function.

Parameters:

- Same parameters as the above *tools.build_sln_command()*.
- **force_vcvars**: Optional. Defaulted to False. Will set `vcvars_command(force=force_vcvars)`.

13.7.7 tools.unzip()

```
def unzip(filename, destination=".", keep_permissions=False)
```

Function mainly used in `source()`, but could be used in `build()` in special cases, as when retrieving pre-built binaries from the Internet.

This function accepts `.tar.gz`, `.tar`, `.tzb2`, `.tar.bz2`, `.tgz` and `.zip` files, and decompress them into the given destination folder (the current one by default).

```
from conans import tools

tools.unzip("myfile.zip")
# or to extract in "myfolder" sub-folder
tools.unzip("myfile.zip", "myfolder")
```

You can keep the permissions of the files using the `keep_permissions=True` parameter.

```
from conans import tools

tools.unzip("myfile.zip", "myfolder", keep_permissions=True)
```

Parameters:

- **filename** (Required): File to be unzipped.
- **destination** (Optional, Defaulted to `"."`): Destination folder for unzipped files.
- **keep_permissions** (Optional, Defaulted to `False`): Keep permissions of files. **WARNING:** Can be dangerous if the zip was not created in a NIX system, the bits could produce undefined permission schema. Use only this option if you are sure that the zip was created correctly.

13.7.8 tools.untargz()

```
def untargz(filename, destination=".")
```

Extract tar gz files (or in the family). This is the function called by the previous `unzip()` for the matching extensions, so generally not needed to be called directly, call `unzip()` instead unless the file had a different extension.

```
from conans import tools

tools.untargz("myfile.tar.gz")
# or to extract in "myfolder" sub-folder
tools.untargz("myfile.tar.gz", "myfolder")
```

Parameters:

- **filename** (Required): File to be unzipped.
- **destination** (Optional, Defaulted to `"."`): Destination folder for *untargzed* files.

13.7.9 tools.get()

```
def get(url, md5="", sha1="", sha256="")
```

Just a high level wrapper for download, unzip, and remove the temporary zip file once unzipped. You can pass hash checking parameters: `md5`, `sha1`, `sha256`. All the specified algorithms will be checked, if any of them doesn't match, it will raise a `ConanException`.

```
from conans import tools

tools.get("http://url/file", md5='d2da0cd0756cd9da6560b9a56016a0cb')
# also, specify a destination folder
tools.get("http://url/file", destination="subfolder")
```

Parameters:

- **url** (Required): URL to download
- **md5** (Optional, Defaulted to `""`): MD5 hash code to check the downloaded file.
- **sha1** (Optional, Defaulted to `""`): SHA1 hash code to check the downloaded file.
- **sha256** (Optional, Defaulted to `""`): SHA256 hash code to check the downloaded file.

13.7.10 tools.get_env()

```
def get_env(env_key, default=None, environment=None)
```

Parses an environment and cast its value against the **default** type passed as an argument.

Following python conventions, returns **default** if **env_key** is not defined.

See an usage example with an environment variable defined while executing conan

```
$ TEST_ENV="1" conan <command> ...
```

```
from conans import tools

tools.get_env("TEST_ENV") # returns "1", returns current value
tools.get_env("TEST_ENV_NOT_DEFINED") # returns None, TEST_ENV_NOT_DEFINED not␣
↪declared
tools.get_env("TEST_ENV_NOT_DEFINED", []) # returns [], TEST_ENV_NOT_DEFINED not␣
↪declared
tools.get_env("TEST_ENV", "2") # returns "1"
tools.get_env("TEST_ENV", False) # returns True (default value is boolean)
tools.get_env("TEST_ENV", 2) # returns 1
tools.get_env("TEST_ENV", 2.0) # returns 1.0
tools.get_env("TEST_ENV", []) # returns ["1"]
```

Parameters:

- **env_key** (Required): environment variable name.
- **default** (Optional, Defaulted to `None`): default value to return if not defined or cast value against.
- **environment** (Optional, Defaulted to `None`): `os.environ` if `None` or environment dictionary to look for.

13.7.11 tools.download()

```
def download(url, filename, verify=True, out=None, retry=2, retry_wait=5,␣
↪overwrite=False,
             auth=None, headers=None)
```

Retrieves a file from a given URL into a file with a given filename. It uses certificates from a list of known verifiers for https downloads, but this can be optionally disabled.

```
from conans import tools

tools.download("http://someurl/somefile.zip", "myfilename.zip")

# to disable verification:
tools.download("http://someurl/somefile.zip", "myfilename.zip", verify=False)

# to retry the download 2 times waiting 5 seconds between them
tools.download("http://someurl/somefile.zip", "myfilename.zip", retry=2, retry_wait=5)

# Use https basic authentication
tools.download("http://someurl/somefile.zip", "myfilename.zip", auth=("user",
↪"password"))

# Pass some header
tools.download("http://someurl/somefile.zip", "myfilename.zip", headers={"Myheader":
↪"My value"})
```

Parameters:

- **url** (Required): URL to download
- **filename** (Required): Name of the file to be created in the local storage
- **verify** (Optional, Defaulted to `True`): When False, disables https certificate validation.
- **out**: (Optional, Defaulted to `None`): An object with a write() method can be passed to get the output, stdout will use if not specified.

- **retry** (Optional, Defaulted to `2`): Number of retries in case of failure.
- **retry_wait** (Optional, Defaulted to `5`): Seconds to wait between download attempts.
- **overwrite**: (Optional, Defaulted to `False`): When *True* Conan will overwrite the destination file if exists, if False it will raise.
- **auth** (Optional, Defaulted to `None`): A tuple of user, password can be passed to use HTTPBasic authentication. This is passed directly to the requests python library, check here other uses of the **auth** parameter: http://docs.python-requests.org/en/master/user/authentication
- **headers** (Optional, Defaulted to `None`): A dict with additional headers.

13.7.12 tools.ftp_download()

```
def ftp_download(ip, filename, login="", password="")
```

Retrieves a file from an FTP server. Right now it doesn't support SSL, but you might implement it yourself using the standard python FTP library, and also if you need some special functionality.

```
from conans import tools

def source(self):
    tools.ftp_download('ftp.debian.org', "debian/README")
    self.output.info(load("README"))
```

Parameters:

- **ip** (Required): The IP or address of the ftp server.
- **filename** (Required): The filename, including the path/folder where it is located.
- **login** (Optional, Defaulted to `""`): Login credentials for the ftp server.
- **password** (Optional, Defaulted to `""`): Password credentials for the ftp server.

13.7.13 tools.replace_in_file()

```
def replace_in_file(file_path, search, replace, strict=True)
```

This function is useful for a simple "patch" or modification of source files. A typical use would be to augment some library existing `CMakeLists.txt` in the `source()` method, so it uses conan dependencies without forking or modifying the original project:

```
from conans import tools

def source(self):
    # get the sources from somewhere
    tools.replace_in_file("hello/CMakeLists.txt", "PROJECT(MyHello)",
        '''PROJECT(MyHello)
           include(${CMAKE_BINARY_DIR}/conanbuildinfo.cmake)
           conan_basic_setup()''')
```

Parameters:

- **file_path** (Required): File path of the file to perform the replace in.
- **search** (Required): String you want to be replaced.

- **replace** (Required): String to replace the searched string.
- **strict** (Optional, Defaulted to `True`): If `True`, it raises an error if the searched string is not found, so nothing is actually replaced.

13.7.14 tools.check_with_algorithm_sum()

```
def check_with_algorithm_sum(algorithm_name, file_path, signature)
```

Useful to check that some downloaded file or resource has a predefined hash, so integrity and security are guaranteed. Something that could be typically done in `source()` method after retrieving some file from the internet.

Parameters:

- **algorithm_name** (Required): Name of the algorithm to be checked.
- **file_path** (Required): File path of the file to be checked.
- **signature** (Required): Hash code that the file should have.

There are specific functions for common algorithms:

```
def check_sha1(file_path, signature)
def check_md5(file_path, signature)
def check_sha256(file_path, signature)
```

For example:

```
from conans import tools

tools.check_sha1("myfile.zip", "eb599ec83d383f0f25691c184f656d40384f9435")
```

Other algorithms are also possible, as long as are recognized by python `hashlib` implementation, via `hashlib.new(algorithm_name)`. The previous is equivalent to:

```
from conans import tools

tools.check_with_algorithm_sum("sha1", "myfile.zip",
                               "eb599ec83d383f0f25691c184f656d40384f9435")
```

13.7.15 tools.patch()

```
def patch(base_path=None, patch_file=None, patch_string=None, strip=0, output=None)
```

Applies a patch from a file or from a string into the given path. The patch should be in diff (unified diff) format. To be used mainly in the `source()` method.

```
from conans import tools

tools.patch(patch_file="file.patch")
# from a string:
patch_content = " real patch content ..."
tools.patch(patch_string=patch_content)
# to apply in subfolder
tools.patch(base_path=mysubfolder, patch_string=patch_content)
```

If the patch to be applied uses alternate paths that have to be stripped, like:

```
--- old_path/text.txt\t2016-01-25 17:57:11.452848309 +0100
+++ new_path/text_new.txt\t2016-01-25 17:57:28.839869950 +0100
@@ -1 +1 @@
- old content
+ new content
```

Then it can be done specifying the number of folders to be stripped from the path:

```
from conans import tools

tools.patch(patch_file="file.patch", strip=1)
```

Parameters:

- **base_path** (Optional, Defaulted to `None`): Base path where the patch should be applied.
- **patch_file** (Optional, Defaulted to `None`): Patch file that should be applied.
- **patch_string** (Optional, Defaulted to `None`): Patch string that should be applied.
- **strip** (Optional, Defaulted to `0`): Number of folders to be stripped from the path.
- **output** (Optional, Defaulted to `None`): Stream object.

13.7.16 tools.environment_append()

```
def environment_append(env_vars)
```

This is a context manager that allows to temporary use environment variables for a specific piece of code in your conanfile:

```
from conans import tools

def build(self):
    with tools.environment_append({"MY_VAR": "3", "CXX": "/path/to/cxx"}):
        do_something()
```

The environment variables will be overridden if the value is a string, while it will be prepended if the value is a list. When the context manager block ends, the environment variables will be unset.

Parameters:

- **env_vars** (Required): Dictionary object with environment variable name and its value.

13.7.17 tools.chdir()

```
def chdir(newdir)
```

This is a context manager that allows to temporary change the current directory in your conanfile:

```
from conans import tools

def build(self):
    with tools.chdir("./subdir"):
        do_something()
```

Parameters:

- **newdir** (Required): Directory path name to change the current directory.

13.7.18 tools.pythonpath()

This tool is automatically applied in the conanfile methods unless *apply_env* is deactivated, so any PYTHONPATH inherited from the requirements will be automatically available.

```
def pythonpath(conanfile)
```

This is a context manager that allows to load the PYTHONPATH for dependent packages, create packages with python code, and reuse that code into your own recipes.

It is automatically applied

```
from conans import tools

def build(self):
    with tools.pythonpath(self):
        from module_name import whatever
        whatever.do_something()
```

When the *apply_env* is activated (default) the above code could be simplified as:

```
from conans import tools

def build(self):
    from module_name import whatever
    whatever.do_something()
```

For that to work, one of the dependencies of the current recipe, must have a `module_name` file or folder with a `whatever` file or object inside, and should have declared in its `package_info()`:

```
from conans import tools

def package_info(self):
    self.env_info.PYTHONPATH.append(self.package_folder)
```

Parameters:

- **conanfile** (Required): Current `ConanFile` object.

13.7.19 tools.no_op()

```
def no_op()
```

Context manager that performs nothing. Useful to condition any other context manager to get a cleaner code:

```
from conans import tools

def build(self):
    with tools.chdir("some_dir") if self.options.myoption else tools.no_op():
        # if not self.options.myoption, we are not in the "some_dir"
        pass
```

13.7.20 tools.human_size()

```
def human_size(size_bytes)
```

Will return a string from a given number of bytes, rounding it to the most appropriate unit: GB, MB, KB, etc. It is mostly used by the conan downloads and unzip progress, but you can use it if you want too.

```
from conans import tools

tools.human_size(1024)
>> 1.0KB
```

Parameters:

- **size_bytes** (Required): Number of bytes.

13.7.21 tools.OSInfo and tools.SystemPackageTool

These are helpers to install system packages. Check *system_requirements()*.

13.7.22 tools.cross_building()

```
def cross_building(settings, self_os=None, self_arch=None)
```

Reading the settings and the current host machine it returns `True` if we are cross building a conan package:

```
from conans import tools

if tools.cross_building(self.settings):
    # Some special action
```

Parameters:

- **settings** (Required): Conanfile settings. Use `self.settings`.
- **self_os** (Optional, Defaulted to `None`): Current operating system where the build is being done.
- **self_arch** (Optional, Defaulted to `None`): Current architecture where the build is being done.

13.7.23 tools.get_gnu_triplet()

```
def get_gnu_triplet(os, arch, compiler=None)
```

Returns string with GNU like `<machine>-<vendor>-<op_system>` triplet.

Parameters:

- **os** (Required): Operating system to be used to create the triplet.
- **arch** (Required): Architecture to be used to create the triplet.
- **compiler** (Optional, Defaulted to `None`): Compiler used to create the triplet (only needed for Windows).

13.7.24 tools.run_in_windows_bash()

```
def run_in_windows_bash(conanfile, bashcmd, cwd=None, subsystem=None, msys_mingw=True,
↪ env=None)
```

Runs an unix command inside a bash shell. It requires to have "bash" in the path. Useful to build libraries using `configure` and `make` in Windows. Check *Windows subsytems* section.

You can customize the path of the bash executable using the environment variable `CONAN_BASH_PATH` or the *conan.conf* `bash_path` variable to change the default bash location.

```
from conans import tools

command = "pwd"
tools.run_in_windows_bash(self, command) # self is a conanfile instance
```

Parameters:

- **conanfile** (Required): Current `ConanFile` object.
- **bashcmd** (Required): String with the command to be run.
- **cwd** (Optional, Defaulted to `None`): Path to directory where to apply the command from.
- **subsystem** (Optional, Defaulted to `None` will autodetect the subsystem). Used to escape the command according to the specified subsystem.
- **msys_mingw** (Optional, Defaulted to `True`) If the specified subsystem is MSYS2, will start it in MinGW mode (native windows development).
- **env** (Optional, Defaulted to `None`) You can pass a dict with environment variable to be applied **at first place** so they will have more priority than others.

13.7.25 tools.get_cased_path()

```
get_cased_path(abs_path)
```

For Windows, for any `abs_path` parameter containing a case-insensitive absolute path, returns it case-sensitive, that is, with the real cased characters. Useful when using Windows subsystems where the file system is case-sensitive.

13.7.26 tools.remove_from_path()

```
remove_from_path(command)
```

This is a context manager that allows you to remove a tool from the PATH. Conan will locate the executable (using `tools.which()`) and will remove from the PATH the directory entry that contains it. It's not necessary to specify the extension.

```
from conans import tools

with tools.remove_from_path("make"):
    self.run("some command")
```

13.7.27 tools.unix_path()

```
def unix_path(path, path_flavor=None)
```

Used to translate Windows paths to MSYS/CYGWIN unix paths like `c/users/path/to/file`.

Parameters:

- **path** (Required): Path to be converted.
- **path_flavor** (Optional, Defaulted to `None`, will try to autodetect the subsystem): Type of unix path to be returned. Options are `MSYS`, `MSYS2`, `CYGWIN`, `WSL` and `SFU`.

13.7.28 tools.escape_windows_cmd()

```
def escape_windows_cmd(command)
```

Useful to escape commands to be executed in a windows bash (msys2, cygwin etc).

- Adds escapes so the argument can be unpacked by `CommandLineToArgvW()`.
- Adds escapes for cmmd.exe so the argument survives cmmd.exe's substitutions.

Parameters:

- **command** (Required): Command to execute.

13.7.29 tools.sha1sum(), sha256sum(), md5sum()

```
def def md5sum(file_path)
def sha1sum(file_path)
def sha256sum(file_path)
```

Return the respective hash or checksum for a file:

```
from conans import tools

md5 = tools.md5sum("myfilepath.txt")
sha1 = tools.sha1sum("myfilepath.txt")
```

Parameters:

- **file_path** (Required): Path to the file.

13.7.30 tools.md5()

```
def md5(content)
```

Returns the MD5 hash for a string or byte object:

```
from conans import tools

md5 = tools.md5("some string, not a file path")
```

Parameters:

- **content** (Required): String or bytes to calculate its md5.

13.7.31 tools.save()

```
def save(path, content, append=False)
```

Utility function to save files in one line. It will manage the open and close of the file and creating directories if necessary.

```
from conans import tools

tools.save("otherfile.txt", "contents of the file")
```

Parameters:

- **path** (Required): Path to the file.
- **content** (Required): Content that should be saved into the file.
- **append** (Optional, Defaulted to `False`): If `True`, it will append the content.

13.7.32 tools.load()

```
def load(path, binary=False)
```

Utility function to load files in one line. It will manage the open and close of the file, and load binary encodings. Returns the content of the file.

```
from conans import tools

content = tools.load("myfile.txt")
```

Parameters:

- **path** (Required): Path to the file.
- **binary** (Optional, Defaulted to `False`): If `True`, it reads the the file as binary code.

13.7.33 tools.mkdir(), tools.rmdir()

```
def mkdir(path)
def rmdir(path)
```

Utility functions to create/delete a directory. The existance of the specified directory is checked, so `mkdir()` will do nothing if the directory already exists and `rmdir()` will do nothing if the directory does not exists.

This makes it safe to use these functions in the `package()` method of a `conanfile.py` when `no_copy_source=True`.

```
from conans import tools

tools.mkdir("mydir") # Creates mydir if it does not already exist
tools.mkdir("mydir") # Does nothing
```

(continues on next page)

(continued from previous page)

```
tools.rmdir("mydir") # Deletes mydir
tools.rmdir("mydir") # Does nothing
```

Parameters:

- **path** (Required): Path to the directory.

13.7.34 tools.which()

```
def which(filename)
```

Returns the path to a specified executable searching in the `PATH` environment variable. If not found, it returns `None`.

This tool also looks for filenames with following extensions if no extension provided:

- `.com`, `.exe`, `.bat` `.cmd` for Windows.
- `.sh` if not Windows.

```
from conans import tools

abs_path_make = tools.which("make")
```

Parameters:

- **filename** (Required): Name of the executable file. It doesn't require the extension of the executable.

13.7.35 tools.touch()

```
def touch(fname, times=None)
```

Updates the timestamp (last access and last modificatiion times) of a file. This is similar to Unix' `touch` command, except the command fails if the file does not exist.

Optionally, a tuple of two numbers can be specified, which denotes the new values for the 'last access' and 'last modified' times respectively.

```
from conans import tools
import time

tools.touch("myfile")                              # Sets atime and mtime to the␣
↪current time
tools.touch("myfile", (time.time(), time.time()) # Similar to above
tools.touch("myfile", (time.time(), 1))          # Modified long, long ago
```

Parameters:

- **fname** (Required): File name of the file to be touched.
- **times** (Optional, Defaulted to `None`: Tuple with 'last access' and 'last modified' times.

13.7.36 tools.relative_dirs()

```
def relative_dirs(path)
```

Recursively walks a given directory (using `os.walk()`) and returns a list of all contained file paths relative to the given directory.

```
from conans import tools

tools.relative_dirs("mydir")
```

Parameters:

- **path** (Required): Path of the directory.

13.7.37 tools.vswhere()

```
def vswhere(all_=False, prerelease=False, products=None, requires=None, version="",
            latest=False, legacy=False, property_="", nologo=True)
```

Wrapper of `vswhere` tool to look for details of Visual Studio installations. Its output is always a list with a dictionary for each installation found.

```
from conans import tools

vs_legacy_installations = tool.vswhere(legacy=True)
```

Parameters:

- **all_** (Optional, Defaulted to `False`): Finds all instances even if they are incomplete and may not launch.
- **prerelease** (Optional, Defaulted to `False`): Also searches prereleases. By default, only releases are searched.
- **products** (Optional, Defaulted to `None`): List of one or more product IDs to find. Defaults to Community, Professional, and Enterprise. Specify `["*"]` by itself to search all product instances installed.
- **requires** (Optional, Defaulted to `None`): List of one or more workload or component IDs required when finding instances. See https://docs.microsoft.com/en-us/visualstudio/install/workload-and-component-ids for a list of workload and component IDs.
- **version** (Optional, Defaulted to `""`): A version range for instances to find. Example: `"[15.0,16.0)"` will find versions 15.*.
- **latest** (Optional, Defaulted to `False`): Return only the newest version and last installed.
- **legacy** (Optional, Defaulted to `False`): Also searches Visual Studio 2015 and older products. Information is limited. This option cannot be used with either `products` or `requires` parameters.
- **property_** (Optional, Defaulted to `""`): The name of a property to return. Use delimiters `.`, `/`, or `_` to separate object and property names. Example: `"properties.nickname"` will return the "nickname" property under "properties".
- **nologo** (Optional, Defaulted to `True`): Do not show logo information.

13.7.38 tools.vs_comntools()

```
def vs_comntools(compiler_version)
```

Returns the value of the environment variable `VS<compiler_version>.0COMNTOOLS` for the compiler version indicated.

```
from conans import tools

vs_path = tools.vs_comntools("14")
```

Parameters:

- **compiler_version** (Required): String with the version number: `"14"`, `"12"`...

13.7.39 tools.vs_installation_path()

```
def vs_installation_path(version, preference=None)
```

Returns the Visual Studio installation path for the given version. It uses `tools.vswhere()` and `tool.vs_comntools()`. It will also look for the installation paths following `CONAN_VS_INSTALLATION_PREFERENCE` environment variable or the preference parameter itself. If the tool is not able to return the path it returns `None`.

```
from conans import tools

vs_path_2017 = tools.vs_installation_path("15", preference=["Community", "BuildTools",
↪ "Professional", "Enterprise"])
```

Parameters:

- **version** (Required): Visual Studio version to locate. Valid version numbers are strings: `"10"`, `"11"`, `"12"`, `"13"`, `"14"`, `"15"`...
- **preference** (Optional, Defaulted to `None`): Set to value of `CONAN_VS_INSTALLATION_PREFERENCE` or defaulted to `["Enterprise", "Professional", "Community", "BuildTools"]`. If only set to one type of preference, it will return the installation path only for that Visual type and version, otherwise `None`.

13.7.40 tools.replace_prefix_in_pc_file()

```
def replace_prefix_in_pc_file(pc_file, new_prefix)
```

Replaces the `prefix` variable in a package config file `.pc` with the specified value.

```
from conans import tools

lib_b_path = self.deps_cpp_info["libB"].rootpath
tools.replace_prefix_in_pc_file("libB.pc", lib_b_path)
```

Parameters:

- **pc_file** (Required): Path to the pc file
- **new_prefix** (Required): New prefix variable value (Usually a path pointing to a package).

See also:

Check section integrations/*pkg-config and pc files* to know more.

13.7.41 tools.collect_libs()

```
def collect_libs(conanfile, folder="lib")
```

Fetches a list of all libraries in the package folder. Useful to collect not inter-dependent libraries or with complex names like `libmylib-x86-debug-en.lib`.

```
from conans import tools

def package_info(self):
    self.cpp_info.libs = tools.collect_libs(self)
```

Parameters:

- **conanfile** (Required): A *ConanFile* object from which to get the *package_folder*.
- **folder** (Optional, Defaulted to `"lib"`): The subfolder where the library files are.

Warning: This tool collects the libraries searching directly inside the package folder and returns them in no specific order. If libraries are inter-dependent, then package_info() method should order them to achieve correct linking order.

13.7.42 tools.PkgConfig()

```
class PkgConfig(object):

    def __init__(self, library, pkg_config_executable="pkg-config", static=False,␣
↪msvc_syntax=False, variables=None)
```

Wrapper of the `pkg-config` tool.

```
from conans import tools

with environment_append({'PKG_CONFIG_PATH': tmp_dir}):
    pkg_config = PkgConfig("libastral")
    print(pkg_config.cflags)
    print(pkg_config.cflags_only_I)
    print(pkg_config.variables)
```

Parameters of the constructor:

- **library** (Required): Library (package) name, such as `libastral`.
- **pkg_config_executable** (Optional, Defaulted to `"pkg-config"`): Specify custom pkg-config executable (e.g. for cross-compilation).
- **static** (Optional, Defaulted to `False`): Output libraries suitable for static linking (adds `--static` to `pkg-config` command line).
- **msvc_syntax** (Optional, Defaulted to `False`): MSVC compatibility (adds `--msvc-syntax` to `pkg-config` command line).
- **variables** (Optional, Defaulted to `None`): Dictionary of pkg-config variables (passed as `--define-variable=VARIABLENAME=VARIABLEVALUE`).

Properties:

PROPERTY	DESCRIPTION
.cflags	get all pre-processor and compiler flags
.cflags_only_I	get -I flags
.cflags_only_other	get cflags not covered by the cflags-only-I option
.libs	get all linker flags
.libs_only_L	get -L flags
.libs_only_l	get -l flags
.libs_only_other	get other libs (e.g. -pthread)
.provides	get which packages the package provides
.requires	get which packages the package requires
.requires_private	get packages the package requires for static linking
.variables	get list of variables defined by the module

13.8 Configuration files

These are the most important configuration files, used to customize conan.

13.8.1 conan.conf

The typical location of the **conan.conf** file is the directory `~/.conan/`:

```
[log]
run_to_output = True        # environment CONAN_LOG_RUN_TO_OUTPUT
run_to_file = False         # environment CONAN_LOG_RUN_TO_FILE
level = 50                  # environment CONAN_LOGGING_LEVEL
# trace_file =              # environment CONAN_TRACE_FILE
print_run_commands = False  # environment CONAN_PRINT_RUN_COMMANDS

[general]
default_profile = default
compression_level = 9                 # environment CONAN_COMPRESSION_LEVEL
sysrequires_sudo = True               # environment CONAN_SYSREQUIRES_SUDO
request_timeout = 30                  # environment CONAN_REQUEST_TIMEOUT
# sysrequires_mode = enabled            # environment CONAN_SYSREQUIRES_MODE (allowed␣
↪modes enabled/verify/disabled)
# vs_installation_preference = Enterprise, Professional, Community, BuildTools #␣
↪environment CONAN_VS_INSTALLATION_PREFERENCE
# verbose_traceback = False           # environment CONAN_VERBOSE_TRACEBACK
# bash_path = ""                      # environment CONAN_BASH_PATH (only windows)
# recipe_linter = False               # environment CONAN_RECIPE_LINTER
# read_only_cache = True              # environment CONAN_READ_ONLY_CACHE
# pylintrc = path/to/pylintrc_file    # environment CONAN_PYLINTRC
# cache_no_locks = True
# user_home_short = your_path         # environment CONAN_USER_HOME_SHORT
# skip_vs_projects_upgrade = False    # environment CONAN_SKIP_VS_PROJECTS_UPGRADE
# non_interactive = False             # environment CONAN_NON_INTERACTIVE

# conan_make_program = make           # environment CONAN_MAKE_PROGRAM (overrides the␣
↪make program used in AutoToolsBuildEnvironment.make)
```

(continues on next page)

(continued from previous page)

```
# cmake_generator                       # environment CONAN_CMAKE_GENERATOR
# http://www.vtk.org/Wiki/CMake_Cross_Compiling
# cmake_toolchain_file                  # environment CONAN_CMAKE_TOOLCHAIN_FILE
# cmake_system_name                     # environment CONAN_CMAKE_SYSTEM_NAME
# cmake_system_version                  # environment CONAN_CMAKE_SYSTEM_VERSION
# cmake_system_processor                # environment CONAN_CMAKE_SYSTEM_PROCESSOR
# cmake_find_root_path                  # environment CONAN_CMAKE_FIND_ROOT_PATH
# cmake_find_root_path_mode_program   # environment CONAN_CMAKE_FIND_ROOT_PATH_MODE_
↪PROGRAM
# cmake_find_root_path_mode_library   # environment CONAN_CMAKE_FIND_ROOT_PATH_MODE_
↪LIBRARY
# cmake_find_root_path_mode_include   # environment CONAN_CMAKE_FIND_ROOT_PATH_MODE_
↪INCLUDE

# cpu_count = 1             # environment CONAN_CPU_COUNT

# Change the default location for building test packages to a temporary folder
# which is deleted after the test (Defaulted to False).
# temp_test_folder = True             # environment CONAN_TEMP_TEST_FOLDER

[storage]
# This is the default path, but you can write your own. It must be an absolute path␣
↪or a
# path beginning with "~" (if the environment var CONAN_USER_HOME is specified, this␣
↪directory, even
# with "~/", will be relative to the conan user home, not to the system user home)
path = ~/.conan/data

[proxies]
# Empty section will try to use system proxies.
# If don't want proxy at all, remove section [proxies]
# As documented in http://docs.python-requests.org/en/latest/user/advanced/#proxies
# http = http://user:pass@10.10.1.10:3128/
# http = http://10.10.1.10:3128
# https = http://10.10.1.10:1080


# Default settings now declared in the default profile
```

Log

The `level` variable, defaulted to 50 (critical events), declares the LOG level . If you want to show more detailed logging information, set this variable to lower values, as 10 to show debug information. You can also adjust the environment variable `CONAN_LOGGING_LEVEL`.

The `print_run_commands`, when is 1, Conan will print the executed commands in `self.run` to the output. You can also adjust the environment variable CONAN_PRINT_RUN_COMMANDS

The `run_to_file` variable, defaulted to False, will print the output from the `self.run` executions to the path that the variable specifies. You can also adjust the environment variable `CONAN_LOG_RUN_TO_FILE`.

The `run_to_output` variable, defaulted to 1, will print to the `stdout` the output from the `self.run` executions in the conanfile. You can also adjust the environment variable `CONAN_LOG_RUN_TO_OUTPUT`.

The `trace_file` variable enable extra logging information about your conan command executions. Set it with an absolute path to a file. You can also adjust the environment variable `CONAN_TRACE_FILE`.

General

The `vs_installation_preference` variable determines the preference of usage when searching a Visual installation. The order of preference by default is Enterprise, Professional, Community and BuildTools. It can be fixed to just one type of installation like only BuildTools. You can also adjust the environment variable `CONAN_VS_INSTALLATION_PREFERENCE`.

The `verbose_traceback` variable will print the complete traceback when an error occurs in a recipe or even in the conan code base, allowing to debug the detected error.

The `bash_path` variable is used only in windows to help the *tools.run_in_windows_bash()* function to locate our Cygwin/MSYS2 bash. Set it with the bash executable path if it's not in the PATH or you want to use a different one.

The `cmake_***` variables will declare the corresponding CMake variable when you use the *cmake generator* and the *CMake build tool*.

The `cpu_count` variable set the number of cores that the *tools.cpu_count()* will return, by default the number of cores available in your machine. Conan recipes can use the cpu_count() tool to build the library using more than one core.

The `pylintrc` variable points to a custom `pylintrc` file that allows configuring custom rules for the python linter executed at `export` time. A use case could be to define some custom indents (though the standard pep8 4-spaces indent is recommended, there are companies that define different styles). The `pylintrc` file has the form:

```
[FORMAT]
indent-string='  '
```

Running `pylint --generate-rcfile` will output a complete rcfile with commments explaining the fields.

The `recipe_linter` variable allows to disable the package recipe analysis (linting) executed at **`conan install`**. Please note that this linting is very recommended, specially for sharing package recipes and collaborating with others.

The `sysrequires_mode` variable, defaulted to `enabled` (allowed modes `enabled/verify/disabled`) controls whether system packages should be installed into the system via `SystemPackageTool` helper, typically used in *system_requirements()*. You can also adjust the environment variable `CONAN_SYSREQUIRES_MODE`.

The `sysrequires_sudo` variable, defaulted to True, controls whether `sudo` is used for installing apt, yum, etc. system packages via `SystemPackageTool`. You can also adjust the environment variable `CONAN_SYSREQUIRES_SUDO`.

The `request_timeout` variable, defaulted to 30 seconds, controls the time after Conan will stop waiting for a response. Timeout is not a time limit on the entire response download; rather, an exception is raised if the server has not issued a response for timeout seconds (more precisely, if no bytes have been received on the underlying socket for timeout seconds). If no timeout is specified explicitly, it do not timeout.

The `user_home_short` specify the base folder to be used with the *short paths* feature. If not specified, the packages marked as *short_paths* will be stored in the *C:.conan* (or the current drive letter).

If the variable is set to "None" will disable the *short_paths* feature in Windows, for modern Windows that enable long paths at the system level.

The `verbose_traceback` variable will print the complete traceback when an error occurs in a recipe or even in the conan code base, allowing to debug the detected error.

Storage

The `storage.path` variable define the path where all the packages will be stored.

On Windows:

- It is recommended to assign it to some unit, e.g. map it to X: in order to avoid hitting the 260 chars path name length limit).
- Also see the *short_paths docs* to know more about how to mitigate the limitation of 260 chars path name length limit.
- It is recommended to disable the Windows indexer or exclude the storage path to avoid problems (busy resources).

Note: If you want to change the default "conan home" (directory where `conan.conf` file is) you can adjust the environment variable `CONAN_USER_HOME`.

Proxies

If you are not using proxies at all, or you want to use the proxies specified by the operating system, just remove the `[proxies]` section completely. You can run **`conan config rm proxies`**.

If you leave leave the `[proxies]` section blank, conan will copy the system configured proxies, but if you configured some exclusion rule it won't work:

```
[proxies]
# Empty section will try to use system proxies.
# If you don't want Conan to mess with proxies at all, remove section [proxies]
```

You can specify http and https proxies as follows. Use the *no_proxy_match* keyword to specify a list of URLs or patterns that will skip the proxy:

```
[proxies]
# As documented in http://docs.python-requests.org/en/latest/user/advanced/#proxies
http: http://user:pass@10.10.1.10:3128/
http: http://10.10.1.10:3128
https: http://10.10.1.10:1080
no_proxy_match: http://url1, http://url2, https://url3*, https://*.custom_domain.*
```

Use *http=None* and/or *https=None* to disable the usage of a proxy.

If this fails, you might also try to set environment variables:

```
# linux/osx
$ export HTTP_PROXY="http://10.10.1.10:3128"
$ export HTTPS_PROXY="http://10.10.1.10:1080"

# with user/password
$ export HTTP_PROXY="http://user:pass@10.10.1.10:3128/"
$ export HTTPS_PROXY="http://user:pass@10.10.1.10:3128/"

# windows (note, no quotes here)
$ set HTTP_PROXY=http://10.10.1.10:3128
$ set HTTPS_PROXY=http://10.10.1.10:1080
```

13.8.2 profiles/default

This is the typical *~/.conan/profiles/default* file:

```
[build_requires]
[settings]
    os=Macos
    arch=x86_64
    compiler=apple-clang
    compiler.version=8.1
    compiler.libcxx=libc++
    build_type=Release
[options]
[env]
```

The settings defaults are the setting values used whenever you issue a **conan install** command over a *conanfile* in one of your projects. The initial values for these default settings are auto-detected the first time you run a **conan** command.

You can override the default settings using the **-s** parameter in **conan install** and **conan info** commands but when you specify a profile, **conan install --profile gcc48**, the default profile won't be applied, unless you specify it with an `include()` statement:

Listing 10: my_clang_profile

```
include(default)

[settings]
compiler=clang
compiler.version=3.5
compiler.libcxx=libstdc++11

[env]
CC=/usr/bin/clang
CXX=/usr/bin/clang++
```

See also:

Check the section *Mastering conan/Profiles* to read more about this feature.

13.8.3 settings.yml

The `settings` are predefined, so only a few, like "os" or "compiler", are possible. They are defined in your `~/.conan/settings.yml` file. Also, the possible values they can take are restricted in the same file. This is done to ensure matching naming and spelling between users, and settings that commonly make sense to most users. Anyway, you can add/remove/modify those settings and their possible values in the `settings.yml` file, according to your needs, just be sure to share changes with colleagues or consumers of your packages.

If you want to distribute a unified `settings.yml` file you can use the *conan config install command*.

Note: The `settings.yml` file is not perfect nor definitive, surely incomplete. Please send us any suggestion (or better a PR) with settings and values that could make sense for other users.

13.8.4 registry.txt

This file is generally automatically managed, and it has also access via the **conan remote** command but just in case you might need to change it. It contains information about the known remotes and from which remotes are each package retrieved:

```
conan-center https://conan.bintray.com True
local http://localhost:9300 True

Hello/0.1@demo/testing local
```

The first section of the file is listing `remote-name`: `remote-url` `verify_ssl`. Adding, removing or changing those lines, will add, remove or change the respective remote. If verify_ssl, conan client will verify the SSL certificates for that remote server.

The second part of the file contains a list of conan-package-reference: remote-name. This is a reference to which remote was that package retrieved from, which will act also as the default for operations on that package.

Be careful when modifying the remotes, as the information of the packages has to remain consistent, e.g. if removing a remote, all package references referencing that remote has to be removed too.

13.8.5 client.crt / client.key

Conan support client TLS certificates. Create a `client.crt` with the client certificate in the conan home directory (default ~/.conan) and a `client.key` with the private key.

You could also create only the `client.crt` file containing both the certificate and the private key concatenated.

13.8.6 artifacts.properties

This file is used to send custom headers in the PUT requests that **`conan upload`** command does:

.conan/artifacts.properties

```
custom_header1=Value1
custom_header2=45
```

Artifactory users can use this file to set file properties for the uploaded files. The variables should have the prefix `artifact_property`. You can use `;` to set multiple values to a property:

.conan/artifacts.properties

```
artifact_property_build.name=Build1
artifact_property_build.number=23
artifact_property_build.timestamp=1487676992
artifact_property_custom_multiple_var=one;two;three;four
```

13.9 Environment variables

These are the environment variables used to customize conan.

Most of them can be set in the `conan.conf` configuration file (inside your `<userhome>/.conan` folder).

13.9.1 CMAKE RELATED VARIABLES

There are some conan environment variables that will set the equivalent CMake variable using the *cmake generator* and the *CMake build tool*:

Variable	CMake set variable
CONAN_CMAKE_TOOLCHAIN_FILE	CMAKE_TOOLCHAIN_FILE
CONAN_CMAKE_SYSTEM_NAME	CMAKE_SYSTEM_NAME
CONAN_CMAKE_SYSTEM_VERSION	CMAKE_SYSTEM_VERSION
CONAN_CMAKE_SYSTEM_PROCESSOR	CMAKE_SYSTEM_PROCESSOR
CONAN_CMAKE_FIND_ROOT_PATH	CMAKE_FIND_ROOT_PATH
CONAN_CMAKE_FIND_ROOT_PATH_MODE_PROGRAM	CMAKE_FIND_ROOT_PATH_MODE_PROGRAM
CONAN_CMAKE_FIND_ROOT_PATH_MODE_LIBRARY	CMAKE_FIND_ROOT_PATH_MODE_LIBRARY
CONAN_CMAKE_FIND_ROOT_PATH_MODE_INCLUDE	CMAKE_FIND_ROOT_PATH_MODE_INCLUDE

See also:

See CMake cross building wiki

13.9.2 CONAN_BASH_PATH

Defaulted to: Not defined

Used only in windows to help the *tools.run_in_windows_bash()* function to locate our Cygwin/MSYS2 bash. Set it with the bash executable path if it's not in the `PATH` or you want to use a different one.

13.9.3 CONAN_CMAKE_GENERATOR

Conan `CMake` helper class is just a convenience to help to translate conan settings and options into cmake parameters, but you can easily do it yourself, or adapt it.

For some compiler configurations, as `gcc` it will use by default the `Unix Makefiles` cmake generator. Note that this is not a package settings, building it with makefiles or other build system, as Ninja, should lead to the same binary if using appropriately the same underlying compiler settings. So it doesn't make sense to provide a setting or option for this.

So it can be set with the environment variable `CONAN_CMAKE_GENERATOR`. Just set its value to your desired cmake generator (as `Ninja`).

13.9.4 CONAN_COLOR_DARK

Defaulted to: `False`/`0`

Set it to `True`/`1` to use dark colors in the terminal output, instead of light ones. Useful for terminal or consoles with light colors as white, so text is rendered in Blue, Black, Magenta, instead of Yellow, Cyan, White.

13.9.5 CONAN_COLOR_DISPLAY

Defaulted to: Not defined

By default if undefined conan output will use color if a tty is detected.

Set it to `False`/`0` to remove console output colors. Set it to `True`/`1` to force console output colors.

13.9.6 CONAN_COMPRESSION_LEVEL

Defaulted to: `9`

Conan uses `tgz` compression for archives before uploading them to remotes. The default compression level is good and fast enough for most cases, but users with huge packages might want to change it and set `CONAN_COMPRESSION_LEVEL` environment variable to a lower number, which is able to get slightly bigger archives but much better compression speed.

13.9.7 CONAN_CPU_COUNT

Defaulted to: Number of available cores in your machine.

Set the number of cores that the *tools.cpu_count()* will return. Conan recipes can use the cpu_count() tool to build the library using more than one core.

13.9.8 CONAN_NON_INTERACTIVE

Defaulted to: `False`/`0`

This environment variable, if set to `True`/`1`, will prevent interactive prompts. Invocations of Conan commands where an interactive prompt would otherwise appear, will fail instead.

This variable can also be set in `conan.conf` as `non_interactive = True` in the `[general]` section.

13.9.9 CONAN_ENV_XXXX_YYYY

You can override the default settings (located in your `~/.conan/profiles/default` directory) with environment variables.

The `XXXX` is the setting name upper-case, and the `YYYY` (optional) is the sub-setting name.

Examples:

- Override the default compiler:

```
CONAN_ENV_COMPILER = "Visual Studio"
```

- Override the default compiler version:

```
CONAN_ENV_COMPILER_VERSION = "14"
```

- Override the architecture:

```
CONAN_ENV_ARCH = "x86"
```

13.9.10 CONAN_LOG_RUN_TO_FILE

Defaulted to: `0`

If set to `1` will log every `self.run("{Some command}")` command output in a file called `conan_run.log`. That file will be located in the current execution directory, so if we call `self.run` in the conanfile.py's build method, the file will be located in the build folder.

In case we execute `self.run` in our `source()` method, the `conan_run.log` will be created in the source directory, but then conan will copy it to the `build` folder following the regular execution flow. So the `conan_run.log` will contain all the logs from your conanfile.py command executions.

The file can be included in the conan package (for debugging purposes) using the `package` method.

```
def package(self):
    self.copy(pattern="conan_run.log", dst="", keep_path=False)
```

13.9.11 CONAN_LOG_RUN_TO_OUTPUT

Defaulted to: `1`

If set to `0` conan won't print the command output to the stdout. Can be used with `CONAN_LOG_RUN_TO_FILE` set to `1` to log only to file and not printing the output.

13.9.12 CONAN_LOGGING_LEVEL

Defaulted to: `50`

By default conan logging level is only set for critical events. If you want to show more detailed logging information, set this variable to lower values, as `10` to show debug information.

13.9.13 CONAN_LOGIN_USERNAME, CONAN_LOGIN_USERNAME_{REMOTE_NAME}

Defaulted to: Not defined

You can define the username for the authentication process using environment variables. Conan will use a variable **CONAN_LOGIN_USERNAME_{REMOTE_NAME}**, if the variable is not declared Conan will use the variable **CONAN_LOGIN_USERNAME**, if the variable is not declared either, Conan will request to the user to input a username.

These variables are useful for unattended executions like CI servers or automated tasks.

If the remote name contains "-" you have to replace it with "_" in the variable name:

For example: For a remote named "conan-center":

```
SET CONAN_LOGIN_USERNAME_CONAN_CENTER=MyUser
```

13.9.14 CONAN_MAKE_PROGRAM

Defaulted to: Not defined

Specify an alternative `make` program to use with:

- The build helper *AutoToolsBuildEnvironment*. Will invoke the specified executable in the *make* method.
- The build helper *build helper CMake*. By adjusting the CMake variable CMAKE_MAKE_PROGRAM.

For example:

```
CONAN_MAKE_PROGRAM="/path/to/mingw32-make"

# Or only the exe name if it is in the path

CONAN_MAKE_PROGRAM="mingw32-make"
```

13.9.15 CONAN_PASSWORD, CONAN_PASSWORD_{REMOTE_NAME}

Defaulted to: Not defined

You can define the authentication password using environment variables. Conan will use a variable **CONAN_PASSWORD_{REMOTE_NAME}**, if the variable is not declared Conan will use the variable **CONAN_PASSWORD**, if the variable is not declared either, Conan will request to the user to input a password.

These variables are useful for unattended executions like CI servers or automated tasks.

If the remote name contains "-" you have to replace it with "_" in the variable name:

For example: For a remote named "conan-center":

```
SET CONAN_PASSWORD_CONAN_CENTER=Mypassword
```

13.9.16 CONAN_PRINT_RUN_COMMANDS

Defaulted to: `0`

If set to `1`, every `self.run("{Some command}")` call will log the executed command {Some command} to the output.

For example: In the *conanfile.py* file:

```
self.run("cd %s && %s ./configure" % (self.ZIP_FOLDER_NAME, env_line))
```

Will print to the output (stout and/or file):

```
----Running------
> cd zlib-1.2.9 && env LIBS="" LDFLAGS=" -m64   $LDFLAGS" CFLAGS="-mstackrealign -
↪fPIC $CFLAGS -m64  -s -DNDEBUG  " CPPFLAGS="$CPPFLAGS -m64  -s -DNDEBUG  " C_
↪INCLUDE_PATH=$C_INCLUDE_PATH: CPLUS_INCLUDE_PATH=$CPLUS_INCLUDE_PATH: ./configure
-----------------
...
```

13.9.17 CONAN_READ_ONLY_CACHE

Defaulted to: Not defined

This environment variable if defined, will make the conan cache read-only. This could prevent developers to accidentally edit some header of their dependencies while navigating code in their IDEs.

This variable can also be set in `conan.conf` as `read_only_cache = True` in the `[general]` section.

The packages are made read-only in two points: when a package is built from sources, and when a package is retrieved from a remote repository.

The packages are not modified for upload, so users should take that into consideration before uploading packages, as they will be read-only and that could have other side-effects.

Warning: It is not recommended to upload packages directly from developers machines with read-only mode as it could lead to insconsistencies. For better reproducibility we recommend that packages are created and uploaded by CI machines.

13.9.18 CONAN_RUN_TESTS

Defaulted to: Not defined (True/False if defined)

This environment variable (if defined) can be used in `conanfile.py` to enable/disable the tests for a library or application.

It can be used as a convention variable and it's specially useful if a library has unit tests and you are doing *cross building*, the target binary can't be executed in current host machine building the package.

It can be defined in your profile files at `~/.conan/profiles`

```
...
[env]
CONAN_RUN_TESTS=False
```

or declared in command line when invoking **`conan install`** to reduce the variable scope for conan execution

```
$ conan install . -e CONAN_RUN_TEST=0
```

See how to retrieve the value with *tools.get_env()* and check an use case with *a header only with unit tests recipe* while cross building.

See example of build method in `conanfile.py` to enable/disable running tests with CMake:

```
from conans import ConanFile, CMake, tools

class HelloConan(ConanFile):
    name = "Hello"
    version = "0.1"

    def build(self):
        cmake = CMake(self)
        cmake.configure()
        cmake.build()
        if tools.get_env("CONAN_RUN_TESTS", True):
            cmake.test()
```

13.9.19 CONAN_SKIP_VS_PROJECTS_UPGRADE

Defaulted to: `False`/`0`

When set to `True`/`1`, the *build_sln_commmand*, the *msvc_build_command* and the *MSBuild()* build helper, will not call `devenv` command to upgrade the `sln` project, irrespective of the `upgrade_project` parameter value.

13.9.20 CONAN_SYSREQUIRES_MODE

Defaulted to: `enabled` allowed values `enabled`/`verify`/`disabled`

This environment variable controls whether system packages should be installed into the system via `SystemPackageTool` helper, typically used in *system_requirements()*.

See values behaviour:

- `enabled`: Default value and any call to install method of `SystemPackageTool` helper should modify the system packages.
- `verify`: Display a report of system packages to be installed and abort with exception. Useful if you don't want to allow conan to modify your system but you want to get a report of packages to be installed.
- `disabled`: Display a report of system packages that should be installed but continue the conan execution and doesn't install any package in your system. Useful if you want to keep manual control of these dependencies, for example in your development environment.

13.9.21 CONAN_SYSREQUIRES_SUDO

Defaulted to: `True`/`1`

This environment variable controls whether `sudo` is used for installing apt, yum, etc. system packages via `SystemPackageTool` helper, typically used in `system_requirements()`. By default when the environment variable does not exist, "True" is assumed, and `sudo` is automatically prefixed in front of package management commands. If you set this to "False" or "0" `sudo` will not be prefixed in front of the comands, however installation or updates of some packages may fail due to a lack of privilege, depending on the user account Conan is running under.

13.9.22 CONAN_TEMP_TEST_FOLDER

Defaulted to: `False`/`0`

Activating this variable will make build folder of *test_package* to be created in the temporary folder of your machine.

13.9.23 CONAN_TRACE_FILE

Defaulted to: Not defined

If you want extra logging information about your conan command executions, you can enable it by setting the `CONAN_TRACE_FILE` environment variable. Set it with an absolute path to a file.

```
export CONAN_TRACE_FILE=/tmp/conan_trace.log
```

When the conan command is executed, some traces will be appended to the specified file. Each line contains a JSON object. The `_action` field contains the action type, like `COMMAND` for command executions, `EXCEPTION` for errors and `REST_API_CALL` for HTTP calls to a remote.

The logger will append the traces until the `CONAN_TRACE_FILE` variable is unset or pointed to a different file.

See also:

Read more here: *How to log and debug a conan execution*

13.9.24 CONAN_USER, CONAN_CHANNEL

Environment variables commonly used in `test_package` conanfiles, to allow package creation for different users and channel without modifying the code. They are also the environment variables that will be checked when using `self.user` or `self.channel` in `conanfile.py` package recipes in user space, where a user/channel has not been assigned yet (it is assigned when exported in the local cache).

See also:

Read more about it in *user, channel*

13.9.25 CONAN_USER_HOME

Defaulted to: Not defined

Allows defining a custom conan cache directory. Can be useful for concurrent builds under different users in CI, to retrieve and store per-project specific dependencies (useful for deployment, for example).

See also:

Read more about it in *Conan local cache: concurrency, Continuous Integration, isolation*

13.9.26 CONAN_USER_HOME_SHORT

Defaulted to: Not defined

Specify the base folder to be used with the *short paths* feature. When not specified, the packages marked as *short_paths* will be stored in the *C:.conan* (or the current drive letter).

If set to "None", it will disable the *short_paths* feature in Windows for modern Windows that enable long paths at the system level.

Note: Please note that this only works with Python 3.6 and newer.

13.9.27 CONAN_VERBOSE_TRACEBACK

Defaulted to: `0`

When an error is raised in a recipe or even in the conan code base, if set to `1` it will show the complete traceback to ease the debugging.

13.9.28 CONAN_VS_INSTALLATION_PREFERENCE

Defaulted to: `Enterprise, Professional, Community, BuildTools`

This envirnoment variables defines the order of preference when searching for a Visual installation product. This would affect every tool that uses `tools.vs_installation_path()` and will search in the order indicated.

For example:

```
set CONAN_VS_INSTALLATION_PREFERENCE=Enterprise, Professional, Community, BuildTools
```

It can also be used to fix the type of installation you want to use indicating just one product type:

```
set CONAN_VS_INSTALLATION_PREFERENCE=BuildTools
```

CHAPTER

FOURTEEN

VIDEOS AND LINKS

- Packaging C/C++ libraries with Conan. 30 min talk by Théo Delrieu at FOSDEM 2018. Includes AndroidNDK package and cross build to Android
- Introduction to Conan C/C++ package manager. 30 min talk in CppCon 2016.
- Faster Delivery of Large C/C++ Projects with Conan Package Manager and Efficient Continuous Integration. 60 min talk in CppCon 2017.
- Conan.io c++ package manager demo with SFML, by Charl Botha

Do you have your own video, tutorial, blog post that could be useful for other users? Please tell us and we will link it here, or directly send a PR to the docs: https://github.com/conan-io/docs.

CHAPTER

FIFTEEN

FAQ

15.1 General

15.1.1 Is Conan CMake based, or is CMake a requirement?

No. It isn't. Conan is build-system agnostic. Package creators could very well use cmake to create their packages, but you will only need it if you want to build packages from source, or if there are no available precompiled packages for your system/settings. We use CMake extensively in our examples and documentation, but only because it is very convenient and most C/C++ devs are familiar with it.

15.1.2 Is build-system XXXXX supported?

Yes. It is. Conan makes no assumption about the build system. It just wraps any build commands specified by the package creators. There are already some helper methods in code to ease the use of CMake, but similar functions can be very easily added for your favourite build system. Please check out the alternatives explained in *generator packages*

15.1.3 Is my compiler, version, architecture, or setting supported?

Yes. Conan is very general, and does not restrict any configuration at all. However, conan comes with some compilers, versions, architectures, ..., etc. pre-configured in the `~/.conan/settings.yml` file, and you can get an error if using settings not present in that file. Go to *invalid settings* to learn more about it.

15.1.4 Does it run offline?

Yes. It runs offline very well. Package recipes and binary packages are stored in your machine, per user, and so you can start new projects that depend on the same libraries without any Internet connection at all. Packages can be fully created, tested and consumed locally, without needing to upload them anywhere.

15.1.5 Is it possible to install 2 different versions of the same library?

Yes. You can install as many different versions of the same library as you need, and easily switch among them in the same project, or have different projects use different versions simultaneously, and without having to install/uninstall or re-build any of them.

Package binaries are stored per user in (e.g.) `~/.conan/data/Boost/1.59/user/stable/package/{sha_0, sha_1, sha_2...}` with a different SHA signature for every different configuration (debug, release, 32-bit, 64-bit, compiler...). Packages are managed per user, but additionally differentiated by version and channel,

and also by their configuration. So large packages, like Boost, don't have to be compiled or downloaded for every project.

15.1.6 Can I run multiple conan isolated instances (virtual environments) on the same machine?

Yes, conan supports the concept of virtual environments; so it manages all the information (packages, remotes, user credentials, ..., etc.) in different, isolated environments. Check *virtual environments* for more details.

15.1.7 Can I run the conan_server behind a firewall (on-premises)?

Yes. Conan does not require a connection to conan.io site or any other external service at all for its operation. You can install packages from the bintray conan-center repository if you want, test them, and only after approval, upload them to your on-premises server and forget about the original repository. Or you can just get the package recipes, re-build from source on your premises, and then upload the packages to your server.

15.1.8 Can I connect to conan remote servers through a corporate proxy?

Yes, it can be configured in your **~/.conan/conan.conf** configuration file or with some environment variables. Check *proxy configuration* for more details.

15.1.9 Can I create packages for third-party libraries?

Of course, as long as their license allows it.

15.1.10 Can I upload closed source libraries?

Yes. As long as the resulting binary artifact can be distributed freely and free of charge, at least for educational and research purposes, and as long as you comply with all licenses and IP rights of the original authors, as well as the Terms of Service. If you want to distribute your libraries only for your paying customers, please contact us.

15.1.11 Do I always need to specify how to build the package from source?

No. But it is highly recommended. If you want, you can just directly start with the binaries, build elsewhere, and upload them directly. Maybe your `build()` step can download pre-compiled binaries from another source and unzip them, instead of actually compiling from sources.

15.1.12 Does conan use semantic versioning (semver) for dependencies?

It uses a convention by which package dependencies follow semver by default; thus it intelligently avoids recompilation/repackaging if you update upstream minor versions, but will correctly do so if you update major versions upstream. This behavior can be easily configured and changed in the `package_id()` method of your conanfile, and any versioning scheme you desire is supported.

15.2 Using conan

15.2.1 How to package header-only libraries?

Packaging header-only libraries is similar to other packages, make sure to first read and understand the *packaging getting started guide*. The main difference is that the package recipe is typically much simpler. There are different approaches depending if you want conan to run the library unit tests while creating the package or not. Full details *in this how-to*.

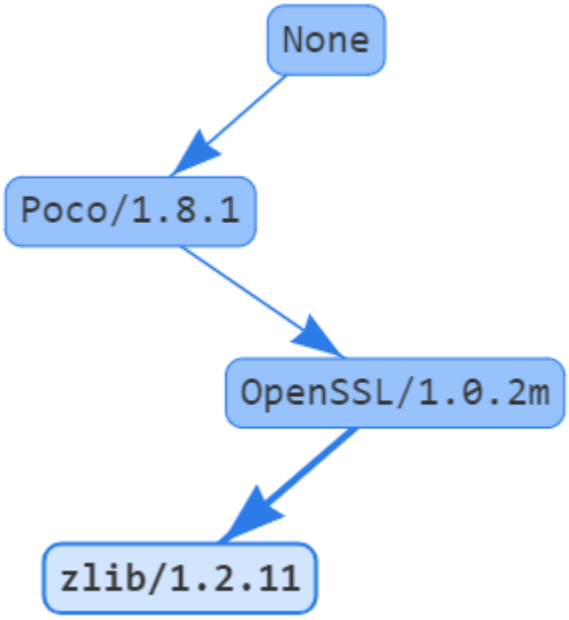
15.2.2 When to use settings or options?

While creating a package you might want to add different configurations and variants of the package. There are 2 main inputs that define packages: settings and options. Read about them in *this section*

15.2.3 How to obtain the dependents of a given package?

The search model for conan in commands such as **`conan install`** and **`conan info`** is done from the downstream or "consumer" package as the starting node of the dependency graph and upstream.

```
$ conan info Poco/1.8.1@pocoproject/stable
```



The inverse model (from upstream to downstream) is not simple to obtain for Conan packages, because the dependency graph is not unique: It changes for every configuration. The graph can be different for different operating systems or just by changing some package options. So you cannot query which packages are dependent on `MyLib/0.1@user/channel`, but which packages are dependent on `MyLib/0.1@user/channel:63da998e3642b50bee33` binary package, and the response can contain many different binary packages for the same recipe, like `MyDependent/0.1@user/channel:packageID1... ID2... MyDependent/0.1@user/channel:packageIDN`. That is the reason why **`conan info`** and **`conan`**

install need a profile (default profile or one given with `--profile`) or installation files `conanbuildinfo.txt` to look for settings and options.

In order to show the inverse graph model, the bottom node is neeed to build the graph upstream and an additonal node too to get the inverse list. This is usually done to get the build order in case a package is updated. For example, if we want to know the build order of the Poco dependecy graph in case OpenSSL is changed we could type:

```
$ conan info Poco/1.8.1@pocoproject/stable -bo OpenSSL/1.0.2m@conan/stable
[OpenSSL/1.0.2m@conan/stable], [Poco/1.8.1@pocoproject/stable]
```

So, if OpenSSL is changed, we would need to rebuild it (of course) and rebuild Poco.

15.3 Troubleshooting

15.3.1 ERROR: Missing prebuilt package

When you are installing packages (with **conan install** or **conan create**) it is possible that you get an error like the following one:

```
WARN: Can't find a 'libzmq/4.2.0@memsharded/testing' package for the specified␣
↪options and settings:
- Settings: arch=x86_64, build_type=Release, compiler=gcc, compiler.libcxx=libstdc++,␣
↪compiler.version=4.9, os=Windows
- Options: shared=False
- Package ID: 7fe67dff831b24bc4a8b5db678a51f1be5e44e7c

ERROR: Missing prebuilt package for 'libzmq/4.2.0@memsharded/testing'
Try to build it from sources with "--build libzmq" or read "http://docs.conan.io/en/
↪latest/faq.html"
```

This means that the package recipe `libzmq/4.2.0@memsharded/testing` exists, but for some reason there is no precompiled package for your current settings. Maybe the package creator didn't build and shared pre-built packages at all and only uploaded the package recipe, or maybe they are only providing packages for some platforms or compilers. E.g. the package creator built packages from the recipe for gcc 4.8 and 4.9, but you are using gcc 5.4.

By default, conan doesn't build packages from sources. There are several possibilities:

- You can try to build the package for your settings from sources, indicating some build policy as argument, like `--build libzmq` or `--build missing`. If the package recipe and the source code work for your settings you will have your binaries built locally and ready for use.
- If building from sources fail, you might want to fork the original recipe, improve it until it supports your configuration, and then use it. Most likely contributing back to the original package creator is the way to go. But you can also upload your modified recipe and pre-built binaries under your own username too.

15.3.2 ERROR: Invalid setting

It might happen sometimes, when you specify a setting not present in the defaults that you receive a message like this:

```
$ conan install -s compiler.version=4.19 ...

ERROR: Invalid setting '4.19' is not a valid 'settings.compiler.version' value.
Possible values are ['4.4', '4.5', '4.6', '4.7', '4.8', '4.9', '5.1', '5.2', '5.3',
↪'5.4', '6.1', '6.2']
Read "http://docs.conan.io/en/latest/faq/troubleshooting.html#error-invalid-setting"
```

This doesn't mean that such architecture is not supported by conan, it is just that it is not present in the actual defaults settings. You can find in your user home folder `~/.conan/settings.yml` a settings file that you can modify, edit, add any setting or any value, with any nesting if necessary.

As long as your team or users have the same settings (you can share with them the file), everything will work. The `settings.yml` file is just a mechanism so users agree on a common spelling for typically settings. Also, if you think that some settings would be useful for many other conan users, please submit it as an issue or a pull request, so it is included in future releases.

It is possible that some build helper, like `CMake` will not understand the new added settings, don't use them or even fail. Such helpers as `CMake` are simple utilities to translate from conan settings to the respective build system syntax and command line arguments, so they can be extended or replaced with your own one that would handle your own private settings.

15.3.3 ERROR: Setting value not defined

When you install or create a package, it is possible to see an error like this:

```
ERROR: Hello/0.1@user/testing: 'settings.arch' value not defined
```

This means that the recipe defined `settings = "os", "arch", ...` but a value for the `arch` setting was not provided either in a profile or in the command line. Make sure to specify a value for it in your profile, or in the command line:

```
$ conan install . -s arch=x86 ...
```

If you are building a pure C library with gcc/clang, you might encounter an error like this:

```
ERROR: Hello/0.1@user/testing: 'settings.compiler.libcxx' value not defined
```

Indeed, for building a C library, it is not necessary to define a C++ standard library. And if you provide a value, you might end with multiple packages for exactly the same binary. What has to be done is to remove such subsetting in your recipe:

```
def configure(self):
    del self.settings.compiler.libcxx
```

15.3.4 ERROR: Failed to create process

When conan is installed via pip/PyPI, and python is installed in a path with spaces (like many times in Windows "C:/Program Files..."), conan can fail to launch. This is a known python issue, and can't be fixed from conan. The current workarounds would be:

- Install python in a path without spaces
- Use virtualenvs. Short guide:

```
$ pip install virtualenvwrapper-win # virtualenvwrapper if not Windows
$ mkvirtualenv conan
(conan) $ pip install conan
(conan) $ conan --help
```

Then, when you will be using conan, for example in a new shell, you have to activate the virtualenv:

```
$ workon conan
(conan) $ conan --help
```

Virtualenvs are very convenient, not only for this workaround, but to keep your system clean and to avoid unwanted interaction between different tools and python projects.

15.3.5 ERROR: Failed to remove folder (Windows)

It is possible that operating conan, some random exceptions (some with complete tracebacks) are produced, related to the impossibility to remove one folder. Two things can happen:

- The user has some file or folder open (in a file editor, in the terminal), so it cannot be removed, and the process fails. Make sure to close files, specially if you are opening or inspecting the local conan cache.
- In Windows, the Search Indexer might be opening and locking the files, producing random, difficult to reproduce and annoying errors. Please **disable the Windows Search Indexer for the conan local storage folder**

CHAPTER

SIXTEEN

CHANGELOG

Check https://github.com/conan-io/conan for issues and more details about development, contributors, etc.

Important: Conan 1.3 shouldn't break any existing 1.0 recipe, or command line invocation. If it does, please report in github. Please read more *about conan stability here*.

16.1 1.3.3 (10-May-2018)

- Bugfix: Fixed encoding issues writing to files and calculating md5 sums.

16.2 1.3.2 (7-May-2018)

- Bugfix: Fixed md5 computation of conan .tgz files for the recipe, exported sources and packages due to file ordering and flags.
- Bugfix: Fixed broken `run_in_windows_bash` due to the wrong argument.
- Bugfix: Fixed `VisualStudioBuildEnvironment` when toolset was not defined.
- Bugfix: Fixed `conan download -p=wrong_id` command
- Fix: Added apple-clang 9.1

16.3 1.3.1 (3-May-2018)

- Bugfix: Fixed regression with `AutoToolsBuildEnvironment` build helper that raised exception with not supported architectures during the calculation of the GNU triplet.
- Bugfix: Fixed `pkg_config` generator, previously crashing when there was no library directories in the requirements.
- Bugfix: Fixed `conanfile.run()` with `win_bash=True` quoting the paths correctly.
- Bugfix: Recovered parameter "append" to the `tools.save` function.
- Bugfix: Added support (documented but missing) to delete options in `package_id()` method using `del self.info.options.<option>`

16.4 1.3.0 (30-April-2018)

- Feature: Added new build types to default `settings.yml`: **RelWithDebInfo** and **MinSizeRel**. Compiler flags will be automatically defined in build helpers that do not understand them (`MSBuild`, `AutotoolsBuildEnvironment`)
- Feature: Improved package integrity. Interrupted downloads or builds shouldn't leave corrupted packages.
- Feature: Added `conan upload --json` json output to the command.
- Feature: new **`conan remove --locks`** to clear cache locks. Useful when killing conan.
- Feature: New **CircleCI** template scripts can be generated with the **`conan new`** command.
- Feature: The CMake() build helper manages the fPIC flag automatically based on the options `fPIC` and `shared` when present.
- Feature: Allowing requiring color output with `CONAN_COLOR_DISPLAY=1` environment variable. If `CONAN_COLOR_DISPLAY` is not set rely on tty detection for colored output
- Feature: New **`conan remote rename`** and **`conan add --force`** commands to handle remotes.
- Feature: Added parameter `use_env` to the `MSBuild().build()` build helper method to control the `/p:UseEnv` msbuild argument.
- Feature: Timeout for downloading files from remotes is now configurable (defaulted to 60 seconds)
- Fix: Use International Units for download/upload transfer sizes (Mb, Kb, etc).
- Fix: Removed duplicated paths in `cmake_multi` generated files.
- Fix: Removed false positive linter warning for local imports.
- Fix: Improved command line help for positional arguments
- Fix **`-ks`** alias for **`--keep-source`** argument in **`conan create`** and **`conan export`**.
- Fix: removed confusing warnings when `self.copy()` doesn't copy files in the `package()` method.
- Fix: `None` is now a possible value for settings with nested subsettings in `settings.yml`.
- Fix: if `vcvars_command` is called and Visual is not found, raise an error instead of warning.
- Bugfix: `self.env_info.paths` and `self.env_info.PATHS` both map now to PATHS env-var.
- Bugfix: Local flow was not correctly recovering state for option values.
- Bugfix: Windows NTFS permissions failed in case USERDOMAIN env-var was not defined.
- Bugfix: Fixed generator `pkg_config` when there are absolute paths (not use prefix)
- Bugfix: Fixed parsing of settings values with `"="` character in conaninfo.txt files.
- Bugfix: Fixed misdetection of MSYS environments (generation of default profile)
- Bugfix: Fixed string scaping in CMake files for preprocessor definitions.
- Bugfix: `upload --no-overwrite` failed when the remote package didn't exist.
- Bugfix: Don't raise an error if `detect_windows_subsystem` doesn't detect a subsystem.

16.5 1.2.3 (10-Apr-2017)

- Bugfix: Removed invalid version field from scons generator.

16.6 1.2.1 (3-Apr-2018)

- Feature: Support for *apple-clang 9.1*
- Bugfix: *compiler_args* generator manage correctly the flag for the *cppstd* setting.
- Bugfix: Replaced exception with a warning message (recommending the *six* module) when using *StringIO* class from the *io* module.

16.7 1.2.0 (28-Mar-2018)

- Feature: The command **conan build** has new `--configure, --build, --install` arguments to control the different stages of the `build()` method.
- Feature: The command **conan export-pkg** now has a **--package-folder** that can be used to export an exact copy of the provided folder, irrespective of the `package()` method. It assumes the package has been locally created with a previous **conan package** or with a **conan build** using a `cmake.install()` or equivalent feature.
- Feature: New `json` generator, generates a json file with machine readable information from dependencies.
- Feature: Improved proxies configuration with `no_proxy_match` configuration variable.
- Feature: New **conan upload** parameter **--no-overwrite** to forbid the overwriting of recipe/packages if they have changed.
- Feature: Exports are now copied to `source_folder` when doing **conan source**.
- Feature: `tools.vcvars()` context manager has no effect if platform is different from Windows.
- Feature: **conan download** has new optional argument **--recipe** to download only the recipe of a package.
- Feature: Added `CONAN_NON_INTERACTIVE` environment variable to disable interactive prompts.
- Feature: Improved `MSbuild()` build helper using `vcvars()` and generating property file to adjust the runtime automatically. New method `get_command()` with the call to `msbuild` tool. Deprecates `tools.build_sln_command()` and `tools.msvc_build_command()`.
- Feature: Support for clang 6.0 correctly managing cppstd flags.
- Feature: Added configuration to specify a client certificate to connect to SSL server.
- Feature: Improved `ycm` generator to show json dependencies.
- Feature: Experimental `--json` parameter for **conan install** and **conan create** to generate a JSON file with install information.
- Fix: **conan install --build** does not absorb more than one parameter.
- Fix: Made conanfile templates generated with **conan new** PEP8 compliant.
- Fix: **conan search** output improved when there are no packages for the given reference.
- Fix: Made **conan download** also retrieve sources.
- Fix: Pylint now runs as an external process.
- Fix: Made `self.user` and `self.channel` available in test_package.
- Fix: Made files writable after a `deploy()` or `imports()` when `CONAN_READ_ONLY_CACHE`/`general.read_only_cache` environment/config variable is `True`.
- Fix: Linter showing warnings with `cpp_info` object in `deploy()` method.

- Fix: Disabled linter for Conan pyinstaller as it was not able to find the python modules.
- Fix: **`conan user -r=remote_name`** showed all users for all remotes, not the one given.
- BugFix: Python reuse code failing to import module in `package_info()`.
- BugFix: Added escapes for backslashes in `cmake` generator.
- BugFix: **`conan config install`** now raises error if **`git clone`** fails.
- BugFix: Alias resolution not working in diamond shaped dependency trees.
- BugFix: Fixed builds with Cygwin/MSYS2 failing in Windows with *self.short_paths=True* and NTFS file systems due to ACL permissions.
- BugFix: Failed to adjust architecture when running Conan platform detection in ARM devices.
- BugFix: Output to StringIO failing in Python 2.
- BugFix: **`conan profile update`** not working to update `[env]` section.
- BugFix: **`conan search`** not creating default remotes when running it as the very first command after Conan installation.
- BugFix: Package folder was not cleaned after the installation and download of a package had failed.

16.8 1.1.1 (5-Mar-2018)

- Feature: `build_sln_command()` and `msvc_build_command()` receive a new optional parameter `platforms` to match the definition of the *.sln* Visual Studio project architecture. (Typically Win32 vs x86 problem).
- Bufix: Flags for Visual Studio command (cl.exe) using "-" instead of "/" to avoid problems in builds using AutoTools scripts with Visual Studio compiler.
- Bugfix: Visual Studio runtime flags adjusted correctly in `AutoToolsBuildEnvironment()` build helper
- Bugfix: `AutoToolsBuildEnvironment()` build helper now adjust the correct build flag, not using eabi suffix, for architecture x86.

16.9 1.1.0 (27-Feb-2018)

- Feature: New **`conan create --keep-build`** option that allows re-packaging from conan local cache, without re-building.
- Feature: **`conan search <pattern> -r=all`** now searches in all defined remotes.
- Feature: Added setting `cppstd` to manage the C++ standard. Also improved build helpers to adjust the standard automatically when the user activates the setting. `AutoToolsBuildEnvironment()`, `CMake()`, `MSBuild()` and `VisualStudioBuildEnvironment()`.
- Feature: New `compiler_args` generator, for directly calling the compiler from command line, for multiple compilers: VS, gcc, clang.
- Feature: Defined `sysrequires_mode` variable (`CONAN_SYSREQUIRES_MODE` env-var) with values `enabled`, `verify`, `disabled` to control the installation of system dependencies via `SystemPackageTool` typically used in *system_requirements()*.
- Feature: automatically apply `pythonpath` environment variable for dependencies containing python code to be reused to recipe `source()`, `build()`, `package()` methods.

- Feature: `CMake` new `patch_config_paths()` methods that will replace absolute paths to conan package path variables, so cmake find scripts are relocatable.
- Feature: new **`--test-build-folder`** command line argument to define the location of the *test_package* build folder, and new conan.conf `temp_test_folder` and environment variable `CONAN_TEMP_TEST_FOLDER`, that if set to True will automatically clean the test_package build folder after running.
- Feature: Conan manages relative urls for upload/download to allow access the server from different configured networks or in domain subdirectories.
- Feature: Added `CONAN_SKIP_VS_PROJECTS_UPGRADE` environment variable to skip the upgrade of Visual Studio project when using *build_sln_commmand*, the *msvc_build_command* and the *MSBuild()* build helper.
- Feature: Improved detection of Visual Studio installations, possible to prioritize between multiple installed Visual tools with the `CONAN_VS_INSTALLATION_PREFERENCE` env-var and `vs_installation_preference` conan.conf variable.
- Feature: Added `keep_path` parameter to `self.copy()` within the `imports()` method.
- Feature: Added `[build_requires]` section to *conanfile.txt*.
- Feature: Added new **`conan help <command>`** command, as an alternative to **`--help`**.
- Feature: Added `target` parameter to `AutoToolsBuildEnvironment.make` method, allowing to select build target on running make
- Feature: The `CONAN_MAKE_PROGRAM` environment variable now it is used by the `CMake()` build helper to set a custom make program.
- Feature: Added **`--verify-ssl`** optional parameter to **`conan config install`** to allow self-signed SSL certificates in download.
- Feature: `tools.get_env()` helper method to automatically convert environment variables to python types.
- Fix: Added a visible warning about `libcxx` compatibility and the detected one for the default profile.
- Fix: Wrong detection of compiler in OSX for gcc frontend to clang.
- Fix: Disabled *conanbuildinfo.cmake* compiler checks for unknown compilers.
- Fix: `visual_studio` generator added missing *ResourceCompile* information.
- Fix: Don't output password from URL for **`conan config install`** command.
- Fix: Signals exit with error code instead of 0.
- Fix: Added package versions to generated SCons file.
- Fix: Error message when package was not found in remotes has been improved.
- Fix: **`conan profile`** help message.
- Fix: Use gcc architecture flags -m32, -m64 for MinGW as well.
- Fix: `CMake()` helper do not require settins if `CONAN_CMAKE_GENERATOR` is defined.
- Fix: improved output of package remote origins.
- Fix: Profiles files use same structure as **`conan profile show`** command.
- Fix: *conanpath.bat* file is removed after conan Windows installer uninstall.
- Fix: Do not add GCC-style flags -m32, -m64, -g, -s to MSVC when using `AutoToolsBuildEnvironment`
- Fix: "Can't find a binary package" message now includes the Package ID.
- Fix: added clang 5.0 and gcc 7.3 to default *settings.yml*.

- Bugfix: `build_id()` logic does not apply unless the `build_id` is effectively changed.
- Bugfix: `self.install_folder` was not correctly set in all necessary cases.
- Bugfix: **`--update`** option does not ignore local packages for version-ranges.
- Bugfix: Set `self.develop=True` for `export-pkg` command.
- Bugfix: Server HTTP responses were incorrectly captured, not showing errors for some server errors.
- Bugfix: Fixed `config` section update for sequential calls over the python API.
- Bugfix: Fixed wrong `self.develop` set to `False` for **`conan create`** with *test_package*.
- Deprecation: Removed **conan-transit** from default remotes registry.

16.10 1.0.4 (30-January-2018)

- Bugfix: Fixed default profile defined in *conan.conf* that includes another profile
- Bugfix: added missing management of `sysroot` in *conanbuildinfo.txt* affecting **`conan build`** and *test_package*.
- Bugfix: Fixed warning in **`conan source`** because of incorrect management of settings.
- Bugfix: Fixed priority order of environment variables defined in included profiles
- Bugfix: NMake error for parallel builds from the `CMake` build helper have been fixed
- Bugfix: Fixed options pattern not applied to root node (`-o *:shared=True` not working for consuming package)
- Bugfix: Fixed shadowed options by package name (`-o *:shared=True -o Pkg:other=False` was not applying `shared` value to Pkg)
- Fix: Using `filter_known_paths=False` as default to `vcvars_dict()` helper.
- Fix: Fixed wrong package name for output messages regarding build-requires
- Fix: Added correct metadata to conan.exe when generated via pyinstaller

16.11 1.0.3 (22-January-2018)

- Bugfix: Correct load of stored settings in conaninfo.txt (for **`conan build`**) when `configure()` remove some setting.
- Bugfix: Correct use of unix paths in Windows subsystems (msys, cygwing) when needed.
- Fix: fixed wrong message for **`conan alias --help`**.
- Fix: Normalized all arguments to **`--xxx-folder`** in command line help.

16.12 1.0.2 (16-January-2018)

- Fix: Adding a warning message for simultaneous use of `os` and `os_build` settings.
- Fix: Do not raise error from *conanbuildinfo.cmake* for Intel MSVC toolsets.
- Fix: Added more architectures to default *settings.yml* `arch_build` setting.

- Fix: using **`--xxx-folder`** in command line help messages.
- Bugfix: using quotes for Windows bash path with spaces.
- Bugfix: vcvars/vcvars_dict not including windows and windows/system32 directories in the path.

16.13 1.0.1 (12-January-2018)

- Fix: **`conan new`** does not generate cross-building (like `os_build`) settings by default. They make only sense for dev-tools used as `build_requires`
- Fix: *conaninfo.txt* file does not dump settings with None values

16.14 1.0.0 (10-January-2018)

- Bugfix: Fixed bug from `remove_from_path` due to Windows path backslash
- Bugfix: Compiler detection in *conanbuildinfo.cmake* for Visual Studio using toolchains like LLVM (Clang)
- Bugfix: Added quotes to bash path.

16.15 1.0.0-beta5 (8-January-2018)

- Fix: Errors from remotes different to a 404 will raise an error. Disconnected remotes have to be removed from remotes or use explicit remote with `-r myremote`
- Fix: cross-building message when building different architecture in same OS
- Fix: **`conan profile show`** now shows profile with same syntax as profile files
- Fix: generated test code in **`conan new`** templates will not run example app if cross building.
- Fix: **`conan export-pkg`** uses the *conanfile.py* folder as the default **`--source-folder`**.
- Bugfix: **`conan download`** didn't download recipe if there are no binaries. Force recipe download.
- Bugfix: Fixed blocked `self.run()` when stderr outputs large tests, due to full pipe.

16.16 1.0.0-beta4 (4-January-2018)

- Feature: `run_in_windows_bash` accepts a dict of environment variables to be prioritised inside the bash shell, mainly intended to control the priority of the tools in the path. Use with `vcvars` context manager and `vcvars_dict`, that returns the PATH environment variable only with the Visual Studio related directories
- Fix: Adding all values to `arch_target`
- Fix: **`conan new`** templates now use new `os_build` and `arch_build` settings
- Fix: Updated `CMake` helper to account for `os_build` and `arch_build` new settings
- Fix: Automatic creation of *default* profile when it is needed by another one (like `include(default)`)
- BugFix: Failed installation (non existing package) was leaving lock files in the cache, reporting a package for **`conan search`**.
- BugFix: Environment variables are now applied to `build_requirements()` for **`conan install ..`**

- BugFix: Dependency graph was raising conflicts for diamonds with **alias** packages.
- BugFix: Fixed **`conan export-pkg`** after a **`conan install`** when recipe has options.

16.17 1.0.0-beta3 (28-December-2017)

- Fix: Upgraded pylint and astroid to latest
- Fix: Fixed `build_requires` with transitive dependencies to other build_requires
- Fix: Improved pyinstaller creation of executable, to allow for py3-64 bits (windows)
- Deprecation: removed all **`--some_argument`**, use instead **`--some-argument`** in command line.

16.18 1.0.0-beta2 (23-December-2017)

- Feature: New command line UI. Most commands use now the path to the package recipe, like **`conan export . user/testing`** or **`conan create folder/myconanfile.py user/channel`**.
- Feature: Better cross-compiling. New settings model for `os_build`, `arch_build`, `os_target`, `arch_target`.
- Feature: Better Windows OSS ecosystem, with utilities and settings model for MSYS, Cygwin, Mingw, WSL
- Feature: `package()` will not warn of not copied files for known use cases.
- Feature: reduce the scope of definition of `cpp_info`, `env_info`, `user_info` attributes to `package_info()` method, to avoid unexpected errors.
- Feature: extended the use of addressing folder and conanfiles with different names for `source`, `package` and `export-pkg` commands
- Feature: added support for Zypper system package tool
- Fix: Fixed application of build requires from profiles that didn't apply to requires in recipes
- Fix: Improved "test package" message in output log
- Fix: updated CI templates generated with **`conan new`**
- Deprecation: Removed `self.copy_headers` and family for the `package()` method
- Deprecation: Removed `self.conanfile_directory` attribute.

Note: This is a beta release, shouldn't be installed unless you do it explicitly

$ pip install conan==1.0.0b2 –upgrade

Breaking changes

- The new command line UI breaks command line tools and integration. Most cases, just add a `.` to the command.
- Removed `self.copy_headers`, `self.copy_libs`, methods for `package()`. Use `self.copy()` instead.
- Removed `self.conanfile_directory` attribute. Use `self.source_folder`, `self.build_folder`, etc. instead

16.19 0.30.3 (15-December-2017)

- Reverted `CMake()` and `Meson()` build helpers to keep old behavior.
- Forced Astroid dependency to < 1.6 because of py3 issues.

16.20 0.30.2 (14-December-2017)

- Fix: `CMake()` and `Meson()` build helpers and relative directories regression.
- Fix: `ycm` generator, removed the access of `cpp_info` to generators, keeping the access to `deps_cpp_info`.

16.21 0.30.1 (12-December-2017)

- Feature: Introduced major versions for gcc (5, 6, 7) as defaults settings for OSS packages, as minors are compatible by default
- Feature: `VisualStudioBuildEnvironment` has added more compilation and link flags.
- Feature: new `MSBuild()` build helper that wraps the call to `msvc_build_command()` with the correct application of environment variables with the improved `VisualStudioBuildEnvironment`
- Feature: `CMake` and `Meson` build helpers got a new `cache_build_dir` argument for `configure(cache_build_dir=None)` that will be used to define a build directory while the package is being built in local cache, but not when built locally
- Feature: `conanfiles` got a new `apply_env` attribute, defaulted to `True`. If false, the environment variables from dependencies will not be automatically applied. Useful if you don't want some dependency adding itself to the PATH by default, for example
- Feature: allow recipes to use and run python code installed with **`conan config install`**.
- Feature: `conanbuildinfo.cmake` now has `KEEP_RPATHS` as argument to keep the RPATHS, as opposed to old SKIP_RPATH which was confusing. Also, it uses set(CMAKE_INSTALL_NAME_DIR "") to keep the old behavior even for CMake >= 3.9
- Feature: **`conan info`** is able to get profile information from the previous install, instead of requiring it as input again
- Feature: `tools.unix_path` support MSYS, Cygwin, WSL path flavors
- Feature: added `destination` folder argument to `tools.get()` function
- Feature: `SystemPackageTool` for apt-get now uses **`--no-install-recommends`** automatically.
- Feature: `visual_studio_multi` generator now uses toolsets instead of IDE version to identify files.
- Fix: generators failures print traces to help debugging
- Fix: typos in generator names, or non-existing generator now raise an Error instead of a warning
- Fix: `short_paths` feature is active by default in Windows. If you want to opt-out, you can use `CONAN_USER_HOME_SHORT=None`
- Fix: `SystemPackageTool` doesn't use sudo in Windows
- BugFix: Not using parallel builds for Visual<10 in CMake build helper.

- Deprecation: `conanfile_directory` shouldn't be used anymore in recipes. Use ``source_folder, build_folder, etc.

Note: Breaking changes

- `scopes` have been completely removed. You can use environment variables, or the `conanfile.develop` or `conanfile.in_local_cache` attributes instead.
- Command *test_package* has been removed. Use **`conan create`** instead, and **`conan test`** for just running package tests.
- `werror` behavior is now by default. Dependencies conflicts will now error, and have to be fixed.
- `short_paths` feature is again active by default in Windows, even with Py3.6 and system LongPathsEnabled.
- `ConfigureEnvironment` and `GCC` build helpers have been completely removed

16.22 0.29.2 (2-December-2017)

- Updated python cryptography requirement for OSX due the pyOpenSSL upgrade. See more: https://pypi.org/project/pyOpenSSL/

16.23 0.29.1 (23-November-2017)

- Support for OSX High Sierra
- Reverted concurrency locks to counters, removed `psutil` dependency
- Implemented migration for settings.yml (for new VS toolsets)
- Fixed encoding issues in conan_server

16.24 0.29.0 (21-November-2017)

- Feature: Support for WindowsStore (WinRT, UWP)
- Feature: Support for Visual Studio Toolsets.
- Feature: New `boost-build` generator for generic bjam (not only Boost)
- Feature: new `tools.PkgConfig` helper to parse pkg-config (.pc) files.
- Feature: Added `self.develop` conanfile variable. It is true for **`conan create`** packages and for local development.
- Feature: Added `self.keep_imports` to avoid removal of imported files in the `build()` method. Convenient for re-packaging.
- Feature: Autodected MSYS2 for `SystemPackageTool`
- Feature: `AutoToolsBuildEnvironment` now auto-loads `pkg_config_path` (to use with `pkg_config` generator)
- Feature: Changed search for profiles. Profiles not found in the default `profiles` folder, will be searched for locally. Use `./myprofile` to force local search only.

- Feature: Parallel builds for Visual Studio (previously it was only parallel compilation within builds)
- Feature: implemented syntax to check options with `if "something" in self.options.myoption`
- Fix: Fixed CMake dependency graph when using TARGETS, that produced wrong link order for transitive dependencies.
- Fix: Trying to download the `exports_sources` is not longer done if such attribute is not defined
- Fix: Added output directories in `cmake` generator for RelWithDebInfo and MinSizeRel configs
- Fix: Locks for concurrent access to local cache now use process IDs (PIDs) to handle interruptions and inconsistent states. Also, adding messages when locking.
- Fix: Not remove the .zip file after a **`conan config install`** if such file is local
- Fix: Fixed `CMake.test()` for the Ninja generator
- Fix: Do not crete local conaninfo.txt file for **`conan install <pkg-ref>`** commands.
- Fix: Solved issue with multiple repetitions of the same command line argument
- BugFix: Don't rebuild conan created (with conan-create) packages when `build_policy="always"`
- BugFix: **`conan copy`** was always copying binaries, now can copy only recipes
- BugFix: A bug in download was causing appends insteads of overwriting for repeated downloads.
- Development: Large restructuring of files (new cmd and build folders)
- Deprecation: Removed old CMake helper methods (only valid constructor is `CMake(self)`)
- Deprecation: Removed old `conan_info()` method, that was superseded by `package_id()`

Note: Breaking changes

- CMAKE_LIBRARY_OUTPUT_DIRECTORY definition has been introduced in `conan_basic_setup()`, it will send shared libraries .so to the `lib` folder in Linux systems. Right now it was undefined.
- Profile search logic has slightly changed. For `-pr=myprofile`, such profile will be searched both in the default folder and in the local one if not existing. Use `-pr=./myprofile` to force local search only.
- The **`conan copy`** command has been fixed. To copy all binaries, it is necessary to explicit **`--all`**, as other commands do.
- The only valid use of CMake helper is `CMake(self)` syntax.
- If using `conan_info()`, replace it with `package_id()`.
- Removed environment variable `CONAN_CMAKE_TOOLSET`, now the toolset can be specified as a subsetting of Visual Studio compiler or specified in the build helpers.

16.25 0.28.1 (31-October-2017)

- BugFix: Downloading (`tools.download`) of files with `content-encoding=gzip` were raising an exception because the downloaded content length didn't match the http header `content-length`

16.26 0.28.0 (26-October-2017)

This is a big release, with many important and core changes. Also with a huge number of community contributions, thanks very much!

- Feature: Major revamp of most conan commands, making command line arguments homogeneous. Much better development flow adapting to user layouts, with `install-folder`, `source-folder`, `build-folder`, `package-folder`.
- Feature: new `deploy()` method, useful for installing binaries from conan packages
- Feature: Implemented some **concurrency** support for the conan local cache. Parallel **`conan install`** and **`conan create`** for different configurations should be possible.
- Feature: options now allow patterns in command line: `-o *:myoption=myvalue` applies to all packages
- Feature: new `pc` generator that generates files from dependencies for `pkg-config`
- Feature: new `Meson` helper, similar to `CMake` for Meson build system. Works well with `pc` generator.
- Feature: Support for read-only cache with `CONAN_READ_ONLY_CACHE` environment variable
- Feature: new `visual_studio_multi` generator to load Debug/Release, 32/64 configs at once
- Feature: new `tools.which` helper to locate executables
- Feature: new **`conan --help`** layout
- Feature: allow to override compiler version in `vcvars_command`
- Feature: **`conan user`** interactive (and not exposed) password input for empty `-p` argument
- Feature: Support for `PacManTool` for `system_requirements()` for ArchLinux
- Feature: Define VS toolset in `CMake` constructor and from environment variable CONAN_CMAKE_TOOLSET
- Feature: **`conan create`** now accepts `werror` argument
- Feature: `AutoToolsBuildEnvironment` can use `CONAN_MAKE_PROGRAM` env-var to define make program
- Feature: added xcode9 for apple-clang 9.0, clang 5 to default settings.yml
- Feature: deactivation of `short_paths` in Windows 10 with Py3.6 and long path support is automatic
- Feature: show unzip progress by percentage, not by file (do not clutters output)
- Feature: do not use `sudo` for system requirements if already running as root
- Feature: `tools.download` able to use headers/auth
- Feature: conan does not longer generate bytecode from recipes (no more .pyc, and more efficient)
- Feature: add parallel argument to `build_sln_command` for VS
- Feature: Show warning if vs150comntools is an invalid path
- Feature: `tools.get()` now has arguments for hash checking
- Fix: upload pattern now accepts `Pkg/*`
- Fix: improved downloader, make more robust, better streaming
- Fix: `tools.patch` now support adding/removal of files
- Fix: The `default` profile is no longer taken as a base and merged with user profile. Use explicit `include(default)` instead.

- Fix: properly manage x86 as cross building with autotools
- Fix: `tools.unzip` removed unnecessary long-paths check in Windows
- Fix: `package_info()` is no longer executed at install for the consumer conanfile.py
- BugFix: source folder was not being correctly removed when recipe was updated
- BugFix: fixed `CMAKE_C_FLAGS_DEBUG` definition in `cmake` generator
- BugFix: `CMAKE_SYSTEM_NAME` is now Darwin for iOS, watchOS and tvOS
- BugFix: `xcode` generator fixed handling of compiler flags
- BugFix: pyinstaller hidden import that broke .deb installer
- BugFix: **`conan profile list`** when local files matched profile names

Note: Breaking changes

This is an important release towards stabilizing conan and moving out of beta. Some breaking changes have been done, but mostly to command line arguments, so they should be easy to fix. Package recipes or existing packages shouldn't break. Please **update**, it is very important to ease the transition of future stable releases. Do not hesitate to ask questions, or for help if you need it. This is a possibly not complete list of things to take into account:

- The command **`conan install`** doesn't accept `cwd` anymore, to change the directory where the generator files are written, use the **`--install-folder`** parameter.
- The command **`conan install`** doesn't accept **`--all`** anymore. Use **`conan download <ref>`** instead.
- The command **`conan build`** now requires the path to the `conanfile.py` (optional before)
- The command **`conan package`** not longer re-package a package in the local cache, now it only operates in a user local folder. The recommended way to re-package a package is using **`conan build`** and then **`conan export-pkg`**.
- Removed **`conan package_files`** in favor of a new command **`conan export-pkg`**. It requires a local recipe with a `package()` method.
- The command **`conan source`** no longer operates in the local cache. now it only operates in a user local folder. If you used **`conan source`** with a reference to workaround the concurrency, now it natively supported, you can remove the command call and trust concurrent install processes.
- The command **`conan imports`** doesn't accept `-d, --dest` anymore, use **`--imports-folder`** parameter instead.
- If you specify a profile in a conan command, like conan create or conan install the base profile *~/.conan/profiles/default* won't be applied. Use explicit `include` to keep the old behavior.

16.27 0.27.0 (20-September-2017)

- Feature: **`conan config install <url>`** new command. Will install remotes, profiles, settings, conan.conf and other files into the local conan installation. Perfect to synchronize configuration among teams
- Feature: improved traceback printing when errors are raised for more context. Configurable via env
- Feature: filtering out non existing directories in `cpp_info` (include, lib, etc), so some build systems don't complain about them.
- Feature: Added include directories to ResourceCompiler and to MIDL compiler in `visual_studio` generator

- Feature: new `visual_studio_legacy` generator for Visual Studio 2008
- Feature: show path where manifests are locally stored
- Feature: `replace_in_file` now raises error if replacement is not done (opt-out parameter)
- Feature: enabled in conan.conf `[proxies]` section `no_proxy=url1,url2` configuration (to skip proxying for those URLs), as well as `http=None` and `https=None` to explicitly disable them.
- Feature: new conanfile `self.in_local_cache` attribute for conditional logic to apply in user folders local commands
- Feature: `CONAN_USER_HOME_SHORT=None` can disable the usage of `short_paths` in Windows, for modern Windows that enable long paths at the system level
- Feature: `if "arm" in self.settings.arch` is now a valid check (without casting to str(self.settings.arch))
- Feature: added cwd`` argument to **`conan source`** local method.
- Fix: unzip crashed for 0 Bytes zip files
- Fix: `collect_libs` moved to the `tools` module
- Bugfix: fixed wrong regex in `deps_cpp_info` causing issues with dots and dashes in package names
- Development: Several internal refactors (tools module, installer), testing (using VS2015 as default, removing VS 12 in testing). Conditional CI in travis for faster builds in developers, downgrading to CMake 3.7 in appveyor
- Deprecation: `dev_requires` have been removed (it was not documented, but accessible via the `requires(dev=True)` parameter. Superseded by `build_requires`.
- Deprecation: sources tgz files for exported sources no longer contain ".c_src" subfolder. Packages created with 0.27 will be incompatible with conan < 0.25

16.28 0.26.1 (05-September-2017)

- Feature: added apple-clang 9.0 to default settings.
- Fix: **`conan copy`** command now supports symlinks.
- Fix: fixed removal of "export_source" folder when files have no permissions
- Bugfix: fixed parsing of *conanbuildinfo.txt* with package names containing dots.

16.29 0.26.0 (31-August-2017)

- Feature: **`conan profile`** command has implemented `update`, `new`, `remove` subcommands, with detect``, to allow creation, edition and management of profiles.
- Feature: **`conan package_files`** command now can call recipe `package()` method if build_folder`` or source_folder`` arguments are defined
- Feature: graph loading algorithm improved to avoid repeating nodes. Results in much faster times for dense graphs, and avoids duplications of private requirements.
- Feature: authentication based on environment variables. Allows very long processes without tokens being expired.
- Feature: Definition of Visual Studio runtime setting `MD` or `MDd` is now automatic based on build type, not necessary to default in profile.

- Feature: Capturing `SystemExit` to return user error codes to the system with `sys.exit(code)`
- Feature: Added SKIP_RPATH argument to cmake `conan_basic_setup()` function
- Feature: Optimized uploads, now uploads will be skipped if there are no changes, irrespective of timestamp
- Feature: Automatic detection of VS 15-2017, via both a `vs150comntools` variable, and using `vswhere.exe`
- Feature: Added NO_OUTPUT_DIRS argument to cmake `conan_basic_setup()` function
- Feature: Add support for Chocolatey system package manager for Windows.
- Feature: Improved in conan user home and path storage configuration, better error checks.
- Feature: `export` command is now able to export recipes without name or version, specifying the full reference.
- Feature: Added new default settings, Arduino, gcc-7.2
- Feature: Add conan settings to cmake generated file
- Feature: new `tools.replace_prefix_in_pc_file()` function to help with .pc files.
- Feature: Adding support for system package tool `pkgutil` on Solaris
- Feature: **`conan remote update`** now allows **`--insert`** argument to change remote order
- Feature: Add `verbose` definition to `CMake` helper.
- Fix: **`conan package`** working locally failed if not specified build_folder
- Fix: Search when using wildcards for version like `Pkg/*@user/channel`
- Fix: Change current working directory to the conanfile.py one before loading it, so relative python imports or code work.
- Fix: `package_files` command now works with `short_paths` too.
- Fix: adding missing require of tested package in test_package/conanfile build() method
- Fix: path joining in `vcvars_command` for custom VS paths defined via env-vars
- Fix: better managing string escaping in CMake variables
- Fix: `ExecutablePath` assignment has been removed from the `visual_studio` generator.
- Fix: removing `export_source` folder containing exported code, fix issues with read-only files and keeps cache consistency better.
- Fix: Accept 100 return code from yum check-update
- Fix: importing *.so files from the **`conan new`** generated test templates
- Fix: progress bars display when download/uploads are not multipart (reported size 0)
- Bugfix: fixed wrong OSX `DYLD_LIBRARY_PATH` variable for virtual environments
- Bugfix: `FileCopier` had a bug that affected `self.copy()` commands, changing base reference directory.

16.30 0.25.1 (20-July-2017)

- Bugfix: Build requires are now applied correctly to test_package projects.
- Fix: Fixed search command to print an error when –table parameter is used without a reference.
- Fix: install() method of the CMake() helper, allows parallel building, change build folder and custom parameters.

- Fix: Controlled errors in migration, print warning if conan is not able to remove a package directory.

16.31 0.25.0 (19-July-2017)

Note: This release introduces a new layout for the local cache, with dedicated `export_source` folder to store the source code exported with `exports_sources` feature, which is much cleaner than the old `.c_src` subfolder. A migration is included to remove from the local cache packages with the old layout.

- Feature: new **`conan create`** command that supersedes *test_package* for creating and testing package. It works even without the test_package folder, and have improved management for user, channel. The test_package recipe no longer defines `requires`
- Feature: new **`conan get`** command that display (with syntax highlight) package recipes, and any other file from conan: recipes, conaninfo.txt, manifests, etc.
- Feature: new **`conan alias`** command that creates a special package recipe, that works like an **alias** or a **proxy** to other package, allowing easy definition and transparent management of "using the latest minor" and similar policies. Those special alias packages do not appear in the dependency graph.
- Feature: new **`conan search --table=file.html`** command that will output an html file with a graphical representation of available binaries
- Feature: created **default profile**, that replace the `[settings_default]` in **conan.conf** and augments it, allowing to define more things like env-vars, options, build_requires, etc.
- Feature: new `self.user_info` member that can be used in `package_info()` to define custom user variables, that will be translated to general purpose variables by generators.
- Feature: **`conan remove`** learned the **`--outdated`** argument, to remove those binary packages that are outdated from the recipe, both from local cache and remotes
- Feature: **`conan search`** learned the **`--outdated`** argument, to show only those binary packages that are outdated from the recipe, both from local cache and remotes
- Feature: Automatic management `CMAKE_TOOLCHAIN_FILE` in `CMake` helper for cross-building.
- Feature: created `conan_api`, a python API interface to conan functionality.
- Feature: new `cmake.install()` method of `CMake` helper.
- Feature: `short_paths` feature now applies also to `exports_sources`
- Feature: `SystemPackageTool` now supports **FreeBSD** system packages
- Feature: `build_requires` now manage options too, also default options in package recipes
- Feature: **`conan build`** learned new **`--package_folder`** argument, useful if the build system perform the packaging
- Feature: `CMake` helper now defines by default `CMAKE_INSTALL_PREFIX` pointing to the current package_folder, so `cmake.install()` can transparently execute the packaging.
- Feature: improved command UX with cwd`` arguments to allow define the current directory for the command
- Feature: improved `VisualStudioBuildEnvironment`
- Feature: transfers now show size (MB, KB) of download/uploaded files, and current status of transfer.
- Feature: **`conan new`** now has arguments to generate CI scripts for Gitlab CI.
- Feature: Added `MinRelSize` and `RelWithDebInfo` management in `CMake` helper.

- Fix: make `mkdir`, `rmdir`, `relative_dirs` available for import from **`conans`** module.
- Fix: improved detection of Visual Studio default under cygwin environment.
- Fix: `package_files` now allows symlinks
- Fix: Windows installer now includes conan_build_info tool.
- Fix: appending environment variables instead of overwriting them when they come from different origins: upstream dependencies and profiles.
- Fix: made opt-in the check of package integrity before uploads, it was taking too much time, and provide little value for most users.
- Fix: Package recipe linter removed some false positives
- Fix: default settings from conan.conf do not fail for constrained settings in recipes.
- Fix: Allowing to define package remote with **`conan remote add_ref`** before download/upload.
- Fix: removed duplicated BUILD_SHARED_LIBS in test_package
- Fix: add "rhel" to list of distros using yum.
- Bugfix: allowing relative paths in `exports` and `exports_sources` fields
- Bugfix: allow custom user generators with underscore

16.32 0.24.0 (15-June-2017)

- Feature: **`conan new`** new arguments to generate **Travis-CI** and **Appveyor** files for Continuous Integration
- Feature: Profile files with `include()` and variable declaration
- Feature: Added `RelWithDebInfo/MinRelSize` to cmake generators
- Feature: Improved linter, removing false positives due to dynamic conanfile attributes
- Feature: Added `tools.ftp_download()` function for FTP retrieval
- Feature: Managing symlinks between folders.
- Feature: **`conan remote add`** command learned new insert`` option to add remotes in specific order.
- Feature: support multi-config in the `SCons` generator
- Feature: support for gcc 7.1+ detection
- Feature: `tools` now are using global `requests` and `output` instances. Proxies will work for `tools.download()`
- Feature: json`` parameter added to **`conan info`** command to create a JSON with the `build_order`.
- Fix: update default repos, now pointing to Bintray.
- Fix: printing `outdated from recipe` also for remotes
- Fix: Fix required slash in `configure_dir` of `AutoToolsBuildEnvironment`
- Fix: command `new` with very short names, now errors earlier.
- Fix: better error detection for incorrect `Conanfile.py` letter case.
- Fix: Improved some cmake robustness using quotes to avoid cmake errors
- BugFix: Fixed incorrect firing of building due to build`` patterns error

- BugFix: Fixed bug with options incorrectly applied to `build_requires` and crashing
- Refactor: internal refactors toward having a python api to conan functionality

16.33 0.23.1 (05-June-2017)

- BugFix: Fixed bug while packaging symlinked folders in build folder, and target not being packaged.
- Relaxed OSX requirement of pyopenssl to <18

16.34 0.23.0 (01-June-2017)

- Feature: new `build_requires` field and `build_requirements()` in package recipes
- Feature: improved commands (source, build, package, package_files) and workflows for local development of packages in user folders.
- Feature: implemented `no_copy_source` attribute in recipes to avoid the copy of source code from "source" to "build folder". Created new `self.source_folder`, `self.build_folder`, `self.package_folder` for recipes to use.
- Feature: improved `qmake` generator with multi-config support, resource directories
- Feature: improved exception capture and formatting for all recipe user methods exceptions
- Feature: new `tools.sha256()` method
- Feature: folder symlinks working now for packages and upload/download
- Feature: added `set_find_paths()` to `cmake-multi`, to set CMake FindXXX.cmake paths. This will work only for single-config build-systems.
- Feature: using environment variables for `configure()`, `requirements()` and `test()` methods
- Feature: added a `pylintrc` environment variable in `conan.conf` to define a PYLINTRC file with custom style definitions (like indents).
- Feature: fixed `vcvars` architecture setting
- Fix: Make `cacert.pem` folder use CONAN_USER_HOME if existing
- Fix: fixed `options=a=b` option definition
- Fix: `package_files` command allows force'' argument to overwrite existing instead of failing
- BugFix: Package names with underscore when parsing `conanbuildinfo.txt`

16.35 0.22.3 (03-May-2017)

- Fix: Fixed CMake generator (in targets mode) with linker/exe flags like –framework XXX containing spaces.

16.36 0.22.2 (20-April-2017)

- Fix: Fixed regression with usernames starting with non-alphabetical characters, introduced by 0.22.0

16.37 0.22.1 (18-April-2017)

- Fix: "-" symbol available again in usernames.
- Fix: Added `future` requirement to solve an error with pyinstaller generating the Windows installer.

16.38 0.22.0 (18-April-2017)

- Feature: `[build_requires]` can now be declared in `profiles` and apply them to build packages. Those requirements are only installed if the package is required to build from sources, and do not affect its package ID hash, and it is not necessary to define them in the package recipe. Ideal for testing libraries, cross compiling toolchains (like Android), development tools, etc.
- Feature: Much improved support for cross-building. Support for cross-building to **Android** provided, with toolchains installable via `build_requires`.
- Feature: New `package_files` command, that is able to create binary packages directly from user files, without needing to define `build()` or `package()` methods in the the recipes.
- Feature: command **`conan new`** with a new bare'' option that will create a minimal package recipe, usable with the `package_files` command.
- Feature: Improved `CMake` helper, with `test()` method, automatic setting of BUILD_SHARED_LIBS, better management of variables, support for parallel compilation in MSVC (via /MP)
- Feature: new `tools.msvc_build_command()` helper that both sets the Visual vcvars and calls Visual to build the solution. Also `vcvars_command` is improved to return non-empty string even if vcvars is set, for easier concatenation.
- Feature: Added package recipe linter, warning for potential errors and also about Python 3 incompatibilities when running from Python 2. Enabled by default can be opt-out.
- Feature: Improvements in HTML output of **`conan info --graph`**.
- Feature: allow custom path to bash, as configuration and environment variable.
- Fix: Not issuing an unused variable warning in CMake for the CONAN_EXPORTED variable
- Fix: added new `mips` architectures and latest compiler versions to default settings.yml
- Fix: Unified username allowed patterns to those used in package references.
- Fix: hardcoded vs15 version in tools.vcvars
- BugFix: Clean crash and improved error messages when manifests mistmatch exists in conan upload.

16.39 0.21.2 (04-April-2017)

- Bugfix: virtualenv generator quoting environment variables in Windows.

16.40 0.21.1 (23-March-2017)

- BugFix: Fixed missing dependencies in `AutoToolsBuildEnvironment`
- BugFix: Escaping single quotes in html graph of **`conan info --graph=file.html`**.

- BugFix: Fixed loading of auth plugins in conan_server
- BugFix: Fixed `visual_studio` generator creating XML with dots.

16.41 0.21.0 (21-March-2017)

- Feature: **`conan info --graph`** or graph=file.html‘‘ will generate a dependency graph representation in dot or html formats.
- Feature: Added better support and tests for Solaris Sparc.
- Feature: custom authenticators are now possible in **`conan_server`**` with plugins.
- Feature: extended **`conan info`** command with path information and filter by packages.
- Feature: enabled conditional binary packages removal with **`conan remove`** with query syntax
- Feature: enabled generation and validation of manifests from *test_package*.
- Feature: allowing `options` definitions in profiles
- Feature: new `RunEnvironment` helper, that makes easier to run binaries from dependent packages
- Feature: new `virtualrunenv` generator that activates environment variable for execution of binaries from installed packages, without requiring `imports` of shared libraries.
- Feature: adding new version modes for ABI compatibility definition in `package_id()`.
- Feature: Extended **`conan new`** command with new option for `exports_sources` example recipe.
- Feature: `CMake` helper defining parallel builds for gcc-like compilers via jN‘‘, allowing user definition with environment variable and in conan.conf.
- Feature: **`conan profile`**` command now show profiles in alphabetical order.
- Feature: extended `visual_studio` generator with more information and binary paths for execution with DLLs paths.
- Feature: Allowing relative paths with $PROFILE_DIR place holder in `profiles`
- Fix: using only file checksums to decide for modified recipe in remote, for possible concurrent builds & uploads.
- Fix: Improved build‘‘ modes management, with better checks and allowing multiple definitions and mixtures of conditions
- Fix: Replaced warning for non-matching OS to one message stating the cross-build
- Fix: local **`conan source`**` command (working in user folder) now properly executes the equivalent of `exports` functionality
- Fix: Setting command line arguments to cmake command as CMake flags, while using the TARGETS approach. Otherwise, arch flags like -m32 -m64 for gcc were not applied.
- BugFix: fixed **`conan imports`** destination folder issue.
- BugFix: Allowing environment variables with spaces
- BugFix: fix for CMake with targets usage of multiple flags.
- BugFix: Fixed crash of `cmake_multi` generator for “multi-config” packages.

16.42 0.20.3 (06-March-2017)

- Fix: Added opt-out for `CMAKE_SYSTEM_NAME` automatically added when cross-building, causing users providing their own cross-build to fail
- BugFix: Corrected usage of `CONAN_CFLAGS` instead of `CONAN_C_FLAGS` in cmake targets

16.43 0.20.2 (02-March-2017)

- Fix: Regression of `visual_studio``generator using ``%(ExecutablePath)` instead of `$(ExecutablePath)`
- Fix: Regression for build=outdated –build=Pkg‘‘ install pattern

16.44 0.20.1 (01-March-2017)

- Fix: Disabled the use of cached settings and options from installed `conaninfo.txt`
- Fix: Revert the use of quotes in `cmake` generator for flags.
- Fix: Allow comments in artifacts.properties
- Fix: Added missing commit for CMake new helpers

16.45 0.20.0 (27-February-2017)

NOTE: It is important that if you upgrade to this version, all the clients connected to the same remote, should upgrade too. Packages created with conan>=0.20.0 might not be usable with conan older conan clients.

- Feature: Largely improved management of **environment variables**, declaration in `package_info()`, definition in profiles, in command line, per package, propagation to consumers.
- Feature: New build helpers `AutotoolsBuildEnvironment`, `VisualStudioBuildEnvironment`, which deprecate `ConfigureEnvironment`, with much better usage of environment variables
- Feature: New `virtualbuildenv` generator that will generate a composable environment with build information from installed dependencies.
- Feature: New `build_id()` recipe method that allows to define logic to build once, and package multiple times without building. E.g.: build once both debug and release artifacts, then package separately.
- Feature: **Multi-config packages**. Now packages can provide multi-configuration packages, like both debug/release artifacts in the same package, with `self.cpp_info.debug.libs = [...]` syntax. Not restricted to debug/release, can be used for other purposes.
- Feature: new **`conan config`** command to manage, edit, display `conan.conf` entries
- Feature: *Improvements* to `CMake` build helper, now it has `configure()` and `build()` methods for common operations.
- Feature: Improvements to `SystemPackageTool` with detection of installed packages, improved implementation, installation of multi-name packages.
- Feature: Unzip with `tools.unzip` maintaining permissions (Linux, OSX)
- Feature: **`conan info`** command now allows profiles too

- Feature: new tools `unix_path()`, `escape_windows_cmd()`, `run_in_windows_bash()`, useful for autotools projects in Win/MinGW/Msys
- Feature: new context manager `tools.chdir`, to temporarily change directory.
- Feature: CMake using `CMAKE_SYSTEM_NAME` for cross-compiling.
- Feature: Artifactory build-info extraction from traces
- Feature: Attach custom headers to artifacts uploads with an *artifacts.properties* file.
- Feature: allow and copy symlinks while **`conan export`**
- Fix: removing quotes in some cmake variables that were generating incorrect builds
- Fix: providing better error messages for non existing binaries, with links to the docs
- Fix: improved error messages if `tools.patch` failed
- Fix: adding `resdirs` to `cpp_info` propagated information, and cmake variables, for directories containing resources and other data.
- Fix: printing error messages if a build‘‘ policy doesn't match any package
- Fix: managing VS2017 by `tools`. Still the manual definition of `vs150comntools` required.
- Bug fix: crashes when not supported characters were dumped to terminal by logger
- Bug fix: wrong executable path in Visual Studio generator

16.46 0.19.3 (27-February-2017)

- Fix: backward compatibility for new environment variables. New features to be introduced in 0.20 will produce that `conaninfo.txt` will not be correctly parsed, and then package would be "missing". This will happen for packages created with 0.20, and consumed with older than 0.19.3

NOTE: It is important that you upgrade at least to this version if you are using remotes with packages that might be created with latest conan releases (like conan.io).

16.47 0.19.2 (15-February-2017)

- Bug fix: Fixed bug with remotes behind proxies
- Bug fix: Fixed bug with `exports_sources` feature and nested folders

16.48 0.19.1 (02-February-2017)

- Bug fix: Fixed issue with **`conan copy`** followed by **`conan upload`** due to the new `exports_sources` feature.

16.49 0.19.0 (31-January-2017)

- Feature: `exports_sources` allows to snapshot sources (like `exports`) but retrieve them strictly when necessary, to build from sources. This can largely improve install times for package recipes containing sources

- Feature: new configurable **tracer** able to create structured logs of conan actions: commands, API calls, etc
- Feature: new logger for `self.run` actions, able to log information from builds and other commands to files, that can afterwards be packaged together with the binaries.
- Feature: support for **Solaris SunOS**
- Feature: `Version` helper improved with `patch`, `pre`, `build` capabilities to handle `1.3.4-alpha2+build1` versions
- Feature: compress level of tgz is now configurable via `CONAN_COMPRESSION_LEVEL` environment variable, default 9. Reducing it can lead to faster compression times, at the expense of slightly bigger archives
- Feature: Add **powershell** support for virtualenv generator in Windows
- Feature: Improved `system_requirements()` raising errors when failing, retrying if not successful, being able to execute in user space for local recipes
- Feature: new cmake helper macro `conan_target_link_libraries()`.
- Feature: new cmake `CONAN_EXPORTED` variable, can be used in CMakeLists.txt to differentiate building in the local conan cache as package and building in user space
- Fix: improving the caching of options from **`conan install`** in conaninfo.txt and precedence.
- Fix: conan definition of cmake output dirs has been disabled for `cmake_multi` generator
- Fix: `imports()` now uses environment variables at "conan install" (but not at "conan imports" yet)
- Fix: `conan_info()` method has been renamed to `package_id()`. Backward compatibility is maintained, but it is strongly encouraged to use the new name.
- Fix: `conan_find_libraries` now use the NO_CMAKE_FIND_ROOT_PATH parameter for avoiding issue while cross-compiling
- Fix: disallowing duplicate URLs in remotes, better error management
- Fix: improved error message for wildcard uploads not matching any package
- Fix: remove deprecated `platform.linux_distribution()`, using new "distro" package
- Bugfix: fixed management of `VerifySSL` parameter for remotes
- Bugfix: fixed misdetection of compiler version in conanbuildinfo.cmake for apple-clang
- Bugfix: fixed trailing slash in remotes URLs producing crashes
- Refactor: A big refactor has been do to `options`. Nested options are no longer supported, and `option.suboption` will be managed as a single string option.

This has been a huge release with contributors of 11 developers. Thanks very much to all of them!

16.50 0.18.1 (11-January-2017)

- Bug Fix: Handling of transitive private dependencies in modern cmake targets
- Bug Fix: Missing quotes in CMake macro for modern cmake targets
- Bug Fix: Handling LINK_FLAGS in cmake modern targets
- Bug Fix: Environment variables no propagating to test project with test_package command

16.51 0.18.0 (3-January-2017)

- Feature: uploads and downloads with **retries** on failures. This helps to avoid having to fully rebuild on CI when a network transfer fails
- Feature: added **SCons** generator
- Feature: support for **Python 3.6**, with several fixes. Added Python 3.6 to CI.
- Feature: show package dates in **`conan info`** command
- Feature: new `cmake_multi` generator for multi-configuration IDEs like Visual Studio and XCode
- Feature: support for **Visual Studio 2017**, VS-15
- Feature: **FreeBSD** now passes test suite
- Feature: **`conan upload`** showing error messages or URL of remote
- Feature: **wildcard or pattern upload**. Useful to upload multiple packages to a remote.
- Feature: allow defining **settings as environment variables**. Useful for use cases like dockerized builds.
- Feature: improved help`` messages
- Feature: cmake helper tools to launch conan directly from cmake
- Added **code coverage** for code repository
- Fix: conan.io badges when containing dash
- Fix: manifests errors due to generated .pyc files
- Bug Fix: unicode error messages crashes
- Bug Fix: duplicated build of same binary package for private dependencies
- Bug Fix: duplicated requirement if using version-ranges and `requirements()` method.

16.52 0.17.2 (21-December-2016)

- Bug Fix: ConfigureEnvironment helper ignoring libcxx setting. #791

16.53 0.17.1 (15-December-2016)

- Bug Fix: conan install –all generating corrupted packages. Thanks to @yogeva
- Improved case sensitive folder management.
- Fix: appveyor links in README.

16.54 0.17.0 (13-December-2016)

- Feature: support for **modern cmake** with cmake `INTERFACE IMPORTED` targets defined per package
- Feature: support for more advanced queries in search.
- Feature: new `profile list|show` command, able to list or show details of profiles

- Feature: adding preliminary support for **FreeBSD**
- Feature: added new `description` field, to document package contents.
- Feature: generation of **imports manifest** and **`conan imports --undo`** functionality to remove imported files
- Feature: optional SSL certificate verification for remotes, to allow self signed certificates
- Feature: allowing custom paths in profiles, so profiles can be easily shared in teams, just inside the source repository or elsewhere.
- Feature: fields `user` and `channel` now available in conan recipes. That allows to declare requirements for the same user/channel as the current package.
- Feature: improved conan.io package web, adding description.
- Fix: allow to modify cmake generator in `CMake` helper class.
- Fix: added `strip` parameter to `tools.patch()` utility
- Fix: removed unused dependency to Boto
- Fix: wrong line endings in Windows for conan.conf
- Fix: proper automatic use of `txt` and `env` generators in *test_package*
- Bug fix: solved problem when uploading python packages that generated .pyc at execution
- Bug fix: crash when duplicate requires were declared in conanfile
- Bug fix: crash with existing imported files with symlinks
- Bug fix: options missing in "copy install command to clipboard" in web

16.55 0.16.1 (05-December-2016)

- Solved bug with *test_package* with arguments, like scopes.

16.56 0.16.0 (19-November-2016)

Upgrade: The build=outdated'' feature had a change in the hash computation, it might report outdated binaries from recipes. You can re-build the binaries or ignore it (if you haven't changed your recipes without re-generating binaries)

- Feature: **version ranges**. Conan now supports defining requirements with version range expressions like `Pkg/[>1.2,<1.9||1.0.1]@user/channel`. Check the *version ranges reference* for details
- Feature: decoupled `imports` from normal install. Now **`conan install --no-imports`** skips the imports section.
- Feature: new **`conan imports`** command that will execute the imports section without running install
- Feature: **overriding settings per package**. Now it is possible to specify individual settings for each package. This can be specified both in the command line and in `profiles`
- Feature: **environment variables** definition in the command line, global and per package. This allows to define specific environment variables as the compiler (CC, CXX) for a specific package. These environment variables can also be defined in `profiles`. Check *profiles reference*
- Feature: Now conan files copies handle **symlinks**, so files are not duplicated. This will save some space and improve download speed in some large packages. To enable it, use `self.copy(..., links=True)`

- Fix: Enabling correct use of **MSYS** in Windows, by using the Windows `C:/...` path instead of the MSYS ones
- Fix: Several fixes in **`conan search`**, both local and in remotes
- Fix: Manifests line endings and order fix, and hash computation fixed (it had wrong ordering)
- Fix: Removed http->https redirection in conan_server that produced some issues for SSL reversed proxies
- Fix: Taking into account "ANY" definition of settings and options
- Fix: Improved some error messages and failures to encode OS errors with unicode characters
- Update: added new arch `ppc64` to default settings
- Update: updated python-requests library version
- Fix: Using `generator()` instead of compiler to decide on cmake multi-configuration for Ninja+cl builds
- Improved and completed documentation

16.57 0.15.0 (08-November-2016)

Upgrade: If you were using the `short_paths` feature in Windows for packages with long paths, please reset your local cache. You could manually remove packages or just run **`conan remove "*"`**

- Feature: New build=outdated`` functionality, that allows to build the binary packages for those dependencies whose recipe has been changed, or if the binary is not existing. Each binary package stores a hash of the recipe to know if they have to be regenerated (are outdated). This information is also provided in the **`conan search <ref>`** command. Useful for package creators and CI.
- Feature: Extended the `short_paths` feature for Windows path limit to the `package` folder, so package with very long paths, typically in headers in nested folder hierarchies are supported.
- Feature: New `tool.build_sln_command()` helper to `build()` Microsoft Visual Studio solution (.sln) projects
- Feature: Extended the `source` and `package` command, so together with `build` they can be fully executed in a user folder, as a convenience for package creation and testing.
- Feature: Extending the scope of `tools.pythonpath` to work in local commands too
- Improved the parsing of `profiles` and better error messages
- Not adding `-s` compiler flag for clang, as it doesn't use it.
- Automatic generation of *conanenv.txt* in local cache, warnings if using local commands and no `conanbuildinfo.txt` and no *conanenv.txt* are present to cache the information form install
- Fix: Fixed bug when using empty initial requirements (`requires = ""`)
- Fix: Added `glob` hidden import to pyinstaller
- Fix: Fixed minor bugs with `short_paths` as local search not listing packages
- Fix: Fixed problem with virtual envs in Windows with paths separator (using / instead of)
- Fix: Fixed parsing of conanbuildinfo.txt, so the root folder for each dependency is available in local commands too
- Fix: Fixed bug in *test_package* with the test project using the `requirements()` method.

16.58 0.14.1 (20-October-2016)

- Fixed bug with *short_paths* feature in windows.
- Improved error messages for non-valid *profile* test files.
- Remove downloaded tgz package files from remotes after decompress them.
- Fixes bug with *install –all* and short_paths

16.59 0.14.0 (20-October-2016)

- Feature: Added profiles, as user predefined settings and environment variables (as CC and CXX for compiler paths). They are stored in files in the conan cache, so they can be easily edited, added, and shared. Use them with **`conan install --profile=name`**
- Feature: `short_paths` feature for Windows now also handle long paths for the final package, in case that a user library has a very long final name, with nested subfolders.
- Feature: Added `tools.cpu_count()` as a helper to retrieve the number of cores, so it can be used in concurrent builds
- Feature: Detects cycles in the dependency graph, and raise error instead of exhausting recursion limits
- Feature: Conan learned the werror‘‘ option that will raise error and stop installation under some cases treated as warnings otherwise: Duplicated dependencies, or dependencies conflicts
- Feature: New `env` generator that generates a text file with the environment variables defined by dependencies, so it can be stored. Such file is parsed by **`conan build`** to be able to use such environment variables for `self.deps_env_info` too, in the same way it uses the `txt` generator to load variables for `self.deps_cpp_info`.
- Fix: Do not print progress bars when output is a file
- Fix: Improved the local conan search, using options too in the query **`conan search -q option=value`**
- Fix: Boto dependency updated to 2.43.0 (necessary for ArchLinux)
- Fix: Simplified the **`conan package`** command, removing unused and confusing options, and more informative messages about errors and utility of this command.
- Fix: More fixes and improvements on `ConfigureEnvironment`, mainly for Windows
- Fix: Conan now does not generate a `conanbuildinfo.txt` file when doing **`conan install <PkgRef>`**.
- Bug fix: Files of a package recipe are “touched” to update their timestamps to current time when retrieved, otherwise some build systems as Ninja can have problems with them.
- Bug fix: `qmake` generator now uses quotes to handle paths with spaces
- Bug fix: Fixed `OSInfo` to return the short distro name instead of the long one.
- Bug fix: fixed transitivy of `private` dependencies

16.60 0.13.3 (13-October-2016)

This minor solves some problems with `ConfigureEnvironment`, mainly for Windows, but also fixes other things:

- Fixed concatenation problems in Windows for several environment variables. Fixed problems with path with spaces
- A batch file is created in Windows to be called, as `if defined` structures doesn't seem to work in the command line.
- The `vcvars_command` from `tools` now checks the Visual Studio environment variable, if it is already set, it will check it with the current project settings, throwing an error if not matching, returning an empty command if matches.
- Added a `compile_flags` property to `ConfigureEnvironment`, to be passed in the command line to the compiler, but not as environment variables
- Added `defines` to environment for nix systems, it was not being handled before
- Added new tests, compiling simple projects and diamond dependencies with cmake, cl (msvc), gcc (gcc in linux, mingw in win) and clang (OSX), for a better coverage of the `ConfigureEnvironment` functionality.
- Fixed wrong `CPP_INCLUDE_PATH`, it is now `CPLUS_INCLUDE_PATH`

16.61 0.13.0 (03-October-2016)

IMPORTANT UPGRADE ISSUE: There was a small error in the computation of binary packages IDs, that has been addressed by conan 0.13. It affects to third level (and higher) binary packages, i.e. A and B in A->B->C->D, which binaries **must** be regenerated for the new hashes. If you don't plan to provide support for older conan releases (<=0.12), which would be reasonable, you should remove all binaries first (**`conan remove -p`**, works both locally and remotely), then re-build your binaries.

Features:

- Streaming from/to disk for all uploads/downloads. Previously, this was done for memory, but conan started to have issues for huge packages (>many hundreds Mbs), that sometimes could be alleviated using Python 64 bits distros. This issues should be alleviated now
- New security system that allows capturing and checking the package recipes and binaries manifests into user folders (project or any other folder). That ensures that packages cannot be replaced, hacked, forged, changed or wrongly edited, either locally or in any remote server, without notice.
- Possible to handle and reuse python code in recipes. Actually, conan can be used as a package manager for python, by adding the package path to `env_info.PYTHONPATH`. Useful if you want to reuse common python code between different package recipes.
- Avoiding re-compress the tgz for packages after uploads if it didn't change.
- New command **`conan source`** that executes the `source()` method of a given conanfile. Very useful for CI, if desired to run in parallel the construction of different binaries.
- New propagation of `cpp_info`, so it now allows for capturing binary packages libraries with new `collect_libs()` helper, and access to created binaries to compute the `package_info()` in general.
- Command *test_package* now allows the update`` option, to automatically update dependencies.
- Added new architectures for `ppc64le` and detection for `AArch64`
- New methods for defining requires effect over binary packages ID (hash) in `conan_info()`
- Many bugs fixes: error in `tools.download` with python 3, restore correct prompt in virtualenvs, bug if removing an option in `config_options()`, setup.py bug...

This release has contributions from @tru, @raulbocanegra, @tivek, @mathieu, and the feedback of many other conan users, thanks very much to all of them!

16.62 0.12.0 (13-September-2016)

- Major changes to **search** api and commands. Decoupled the search of package recipes, from the search of binary packages.
- Fixed bug that didn't allow to `export` or `upload` packages with settings restrictions if the restrictions didn't match the host settings
- Allowing disabling color output with `CONAN_COLOR_DISPLAY=0` environment variable, or to configure color schema for light console backgrounds with `CONAN_COLOR_DARK=1` environment variable
- Imports can use absolute paths, and files copied from local conan cache to those paths will not be removed when **`conan install`**. Can be used as a way to install machine-wise things (outside conan local cache)
- More robust handling of failing transfers (network disconnect), and inconsistent status after such
- Large internal refactor for storage managers. Improved implementations and decoupling between server and client
- Fixed slow **`conan remove`** for caches with many packages due to slow deletion of empty folders
- Always allowing explicit options scopes, `-o Package:option=value` as well as the implicit `-o option=value` for current `Package`, for consistency
- Fixed some bugs in client-server auth process.
- Allow to extract `.tar` files in `tools.unzip()`
- Some helpers for `conan_info()`, as `self.info.requires.clear()` and removal of settings and options

16.63 0.11.1 (31-August-2016)

- New error reporting for failures in conanfiles, including line number and offending line, much easier for package creators
- Removed message requesting to create an account in **conan.io** for other remotes
- Removed localhost:9300 remote that was added by default mostly for demo purposes. Clarified in docs.
- Fixed usernames case-sensitivity in conan_server, due to ConfigParser it was forcing lowercase
- Handling unicode characters in remote responses, fixed crash
- Added new compilers gcc 6.2, clang 8.0 to the default `settings.yml`
- Bumped cryptography, boto and other conan dependencies, mostly for ArchLinux compatibility and new OSX security changes

16.64 0.11.0 (3-August-2016)

- New solution for the path length limit in Windows, more robust and complete. Package conanfile.py just have to declare an attribute `short_paths=True` and everything will be managed. The old approach is deprecated and totally removed, so no shorts_paths.conf file is necessary. It should fix also the issues with uploads/retrievals.
- New `virtualenv` generator that generates `activate` and `deactivate` scripts that set environment variables in the current shell. It is very useful, for example to install tools (like CMake, MinGW) with conan

packages, so multiple versions can be installed in the same machine, and switch between them just by activating such virtual environments. Packages for MinGW and CMake are already available as a demo

- ConfigureEnvironment takes into account environment variables, defined in packages in new `env_info`, which is similar to `cpp_info` but for environment information (like paths).
- New per-package `build_policy`, which can be set to `always` or `missing`, so it is not necessary to create packages or specify the build‘‘ parameter in command line. Useful for example in header only libraries or to create packages that always get the latest code from a branch in a github repository.
- Command **`conan test_package`**` now executes by default a **`conan export`** with smarter package reference deduction. It is introduced as opt-out behavior.
- Conan :command‘export‘ command avoids copying *test_package/build* temporary files in case of `export=*`
- Now, `package_info()` allows absolute paths in `includedir`, `libdirs` and `bindirs`, so wrapper packages can be defined that use system or manually installed libraries.
- LDFLAGS in `ConfigureEnvironment` management of OSX frameworks.
- Options allow the `ANY` value, so such option would accept any value. For example a commit of a git repository, useful to create packages that can build any specific commit of a git repo.
- Added gcc 5.4 to the default settings, as well as MinGW options (Exceptions, threads...)
- Command **`conan info`** learned a new option to output the packages from a project dependency tree that should be rebuilt in case of a modification of a certain package. It outputs a machine readable **ordered** list of packages to be built in that order. Useful for CI systems.
- Better management of incomplete, dirty or failed `source` directories (e.g. in case of a user interrupting with Ctrl+C a git clone inside the `source()` method.
- Added tools for easier detection of different OS versions and distributions, as well as command wrappers to install system packages (apt, yum). They use `sudo` via a new environment variable CONAN_SYSREQUIRES_SUDO, so using sudo is opt-in/out, for users with different sudo needs. Useful for `system_requirements()`
- Deprecated the `config()` method (still works, for backwards compatibility), but has been replaced by a `config_options()` to modify options based on settings, and a `configure()` method for most use cases. This removes a nasty behaviour of having the `config()` method called twice with side effects.
- Now, running a **`conan install MyLib/0.1@user/channel`** to directly install packages without any consuming project, is also able to generate files with the `-g` option. Useful for installing tool packages (MinGW, CMake) and generate `virtualenvs`.
- Many small fixes and improvements: detect compiler bug in Py3, search was crashing for remotes, conan new failed if the package name had a dash, etc.
- Improved some internal duplications of code, refactored many tests.

This has been a big release. Practically 100% of the released features are thanks to active users feedback and contributions. Thanks very much again to all of them!

16.65 0.10.0 (29-June-2016)

- **conan new** command, that creates conan package conanfile.py templates, with a *test_package* package test (-t option), also for header only packages (-i option)
- Definition of **scopes**. There is a default **dev** scope for the user project, but any other scope (test, profile...) can be defined and used in packages. They can be used to fire extra processes (as running tests), but they do not affect the package binares, and are not included in the package IDs (hash).

- Definition of **dev_requires**. Those are requirements that are only retrieved when the package is in **dev** scope, otherwise they are not. They do not affect the binary packages. Typical use cases would be test libraries or build scripts.
- Allow **shorter paths** for specific packages, which can be necessary to build packages with very long path names (e.g. Qt) in Windows.
- Support for bzip2 and gzip decompression in `tools`
- Added `package_folder` attribute to conanfile, so the `package()` method can for example call `cmake install` to create the package.
- Added `CONAN_CMAKE_GENERATOR` environment variable that allows to override the `CMake` default generator. That can be useful to build with Ninja instead of the default Unix Makefiles
- Improved `ConfigureEnvironment` with include paths in CFLAGS and CPPFLAGS, and fixed bug.
- New **`conan user --clean`** option, to completely remove all user data for all remotes.
- Allowed to raise `Exceptions` in `config()` method, so it is easier for package creators to raise under non-supported configurations
- Fixed many small bugs and other small improvements

As always, thanks very much to all contributors and users providing feedback.

16.66 0.9.2 (11-May-2016)

- **Fixed download bug** that made it specially slow to download, even crash. Thanks to github @melmdk for fixing it.
- **Fixed cmake check of CLang**, it was being skipped
- **Improved performance**. Check for updates has been removed from install, made it opt-in in **`conan info`** command, as it was very slow, seriously affecting performance of large projects.
- Improved internal representation of graph, also improves performance for large projects.
- Fixed bug in **`conan install --update`**.

16.67 0.9 (3-May-2016)

- **Python 3** "experimental" support. Now the main conan codebase is Python 2 and 3 compatible. Python 2 still the reference platform, Python 3 stable support in next releases.
- Create and share your **own custom generators for any build system or tool**. With "generator packages", you can write a generator just as any other package, upload it, modify and version it, etc. Require them by reference, as any other package, and pull it into your projects dynamically.
- **Premake4** initial experimental support via a generator package.
- Very large **re-write of the documentation**. New "creating packages" sections with in-source and out-source explicit examples. Please read it! :)
- Improved **`conan test`**. Renamed `test` to *test_package* both for the command and the folder, but backwards compatibility remains. Custom folder name also possible. **Adapted test layout** might require minor changes to your package test, automatic warnings added for your convenience.
- Upgraded pyinstaller to generate binary OS installers from 2.X to 3.1

- **conan search** now has command line options:, less verbose, verbose, extra verbose
- Added variable with full list of dependencies in conanbuildinfo.cmake
- Several minor bugfixes (check github issues)
- Improved **conan user** to manage user login to multiple remotes

16.68 0.8.4 (28-Mar-2016)

- Fixed linker problems with the new apple-clang 7.3 due to libraries with no setted timestamp.
- Added apple-clang 7.3 to default settings
- Fixed default libcxx for apple-clang in auto detection of base conan.conf

16.69 0.8 (15-Mar-2016)

- New **conan remote** command to manage remotes. Redesigned remotes architecture, now allows to work with several remotes in a more consistent, powerful and "git-like" way. New remotes registry keeps track of the remote of every installed package, and this information is shown in **conan info** command too. Also, it keeps different user logins for different remotes, to improve support in corporate environments running in-house servers.
- New **update** functionality. Now it is possible to **conan install --update** to update packages that became obsolete because new ones were uploaded to the corresponding remote. Conan commands as install and info show information about the status of the local packages compared with the remote ones. In this way, using latest versions during development is much more natural.
- Added new **compiler.libcxx** setting in order to support the different c++ standard libraries. It can take libstdc++, libstdc++11 or libc++ values to take into account different standard libraries for modern gcc and clang compilers. It is also possible to remove not needed settings, like this one in pure C projects, with the new syntax: `del self.settings.compiler.libcxx`
- Conan **virtual environment**: Define a custom conan directory with **CONAN_USER_HOME** env variable, and have a per project or per workspace storage for your dependencies. So you can isolate your dependencies and even bundle them within your project, by just setting the CONAN_USER_HOME variable to your `<project>/deps` folder, for example. This also improves support for continuous integration CI systems, in which many builds from different users could be run in parallel.
- Better conanfile download method. More stable and now checks (opt-out) the **ssl certificates**.
- Lots of improvements: Increased library name length limit, Improved and cleaner output messages.
- Fixed several minor bugs: removing empty folders, case sensitive exports, arm settings detection.
- Introduced the concept of **"package recipe"** that refers to conanfile.py and exported files.
- Improved settings display in web, with new "copy install command to clipboard" to assist in installing packages discovered in web.
- The OSX installer, problematic with latest OSX releases, has been deprecated in favour of homebrew and pip install procedures.

16.70 0.7 (5-Feb-2016)

- Custom conanfile names are allowed for developing. With file‘‘ option you can define the file you want to use, allowing for `.conaninfo.txt` or having multiple `conanfile_dev.py`, `conanfile_test.py` besides the standard `conanfile.py` which is used for sharing the package. Inheritance is allowed, e.g. `conanfile_dev.py` might extend/inherit from `conanfile.py`.
- New **`conan copy`** command that can be used to copy/rename packages, promote them between channels, forking other users packages.
- New all‘‘ and package‘‘ options for **`conan install`** that allows to download one, several, or all package configurations for a given reference.
- Added `patch()` tool to easily patch sources if necessary.
- New **qmake** and **qbs** generators
- Upload of conanfile **exported** files is also **tgz'd**, allowing fast upload/downloads of full sources if desired, avoiding retrieval of sources from externals sources.
- **`conan info`** command improved showing info of current project too
- Output of `run()` can be redirected to buffer string for processing, or even removed.
- Added **proxy** configuration to conan.conf for users behinds proxies.
- Large improvements in commands output, prefixed with package reference, and much clear.
- Updated settings for more versions of gcc and new arm architectures
- Treat dependencies includes as SYSTEM in cmake, so no warnings are raised
- Deleting source folder after **`conan export`** so no manual removal is needed
- Normalizing to CRLF generated user files in Win
- Better detection and checks for compilers as VS, apple-clang
- Fixed CMAKE_SHARED_LINKER_FLAGS typo in cmake files
- Large internal refactor in generators

16.71 0.6 (11-Jan-2016)

- New cmake variables in cmake generator to make FindPackage work better thanks to the underlaying FindLibrary. Now many FindXXX.cmake work “as-is” and the package creator does not have to create a custom override, and consumers can use packages transparently with the originals FindXXX.cmakes
- New “conan info” command that shows the full dependency graph and details (license, author, url, dependants, dependencies) for each dependency.
- New environment helper with a ConfigureEnvironment class, that is able to translate conan information to autotools configure environment definition
- Relative importing from conanfiles now is possible. So if you have common functionality between different packages, you can reuse those python files by importing them from the conanfile.py. Note that export=”...” might be necessary, as packages as to be self-contained.
- Added YouCompleteMe generator for vim auto-completion of dependencies.

- New "conanfile_directory" property that points to the file in which the conanfile.py is located. This helps if using the conanfile.py "build" method to build your own project as a project, not a package, to be able to use any workflow, out-of-source builds, etc.
- Many edits and improvements in help, docs, output messages for many commands.
- All cmake syntax in modern lowercase
- Fixed several minor bugs: gcc detection failure when gcc not installed, missing import, copying source->build failing when symlinks

16.72 0.5 (18-Dec-2015)

- New cmake functionality allows package creators to provide cmake finders, so that package consumers can use their CMakeLists.txt with typical FindXXX.cmake files, without any change to them. CMake CONAN_CMAKE_MODULES_PATH added, so that package creators can provide any additional cmake scripts for consumers.
- Now it is possible to generate out-of-source and multiple configuration installations for the same project, so you can switch between them without having to **`conan install`** again. Check *the new workflows*
- New qmake generator (thanks @dragly)
- Improved removal/deletion of folders with shutil.rmtree, so **`conan remove`** commands and other processes requiring deletion of folders do not fail due to permissions and require manual deletion. This is an improvement, especially in Win.
- Created `pip` package, so conan can be installed via: `pip install conan`
- Released `pyinstaller` code for the creation of binaries from conan python source code. Distros package creators can create packages for the conan apps easily from those binaries.
- Added md5, sha1, sha256 helpers in `tools`, so external downloads from `conanfile.py` files `source()` can be checked.
- Added latest gcc versions to default `settings.yml`
- Added CI support for conan development: travis-ci, appveyor
- Improved human-readability for download progress, help messages.
- Minor bug fixes