Conan Documentation

Release 2.0.0-alpha

The Conan team

Mar 21, 2022

CONTENTS

CHAPTER

ONE

INTRODUCTION

CHAPTER TWO

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. Note that some of **these installers might have some limitations**, especially those created with pyinstaller (such as Windows exe & Linux deb).
3. Running Conan from sources.

2.1 Install with pip (recommended)

To install latest Conan 2.0 pre-release version using `pip`, you need a Python >= 3.6 distribution installed on 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.x and Python <= 3.5 support has been dropped. Conan will not work with those python versions.

Install Conan:

```
$ pip install conan --pre
```

Important: Please READ carefully

- Make sure that your **pip** installation matches your **Python (>= 3.6)** version.
- In **Linux**, you may need **sudo** permissions to install Conan globally.
- We strongly recommend using **virtualenvs** (virtualenvwrapper works great) for everything related to Python. (check https://virtualenvwrapper.readthedocs.io/en/stable/, or https://pypi.org/project/virtualenvwrapper-win/ in Windows) With Python 3, the built-in module `venv` can also be used instead (check https://docs.python.org/3/library/venv.html). If not using a **virtualenv** it is possible that conan dependencies will conflict with previously existing dependencies, especially if you are using Python for other purposes.
- In **OSX**, especially the latest versions that may have **System Integrity Protection**, pip may fail. Try using virtualenvs, or install with another user `$ pip install --user conan`.

- Some Linux distros, such as Linux Mint, require a restart (shell restart, or logout/system if not enough) after installation, so Conan is found in the path.
- In Windows, Python 3 installation can fail installing the `wrapt` dependency because of a bug in **pip**. Information about this issue and workarounds is available here: https://github.com/GrahamDumpleton/wrapt/issues/112.

2.1.1 Known installation issues with pip

- When Conan is installed with **`pip install --user <username>`**, usually a new directory is created for it. However, the directory is not appended automatically to the *PATH* and the **`conan`** commands do not work. This can usually be solved restarting the session of the terminal or running the following command:

  ```
  $ source ~/.profile
  ```

2.2 Install from source

You can run Conan directly from source code. First, you need to install Python and pip.

Clone (or download and unzip) the git repository and install it.

Conan 2 is still in alpha stage, so you must check the *develop2* branch of the repository:

```
# clone folder name matters, to avoid imports issues
$ git clone https://github.com/conan-io/conan.git conan_src
$ cd conan_src
$ git fetch --all
$ git checkout -b develop2 origin/develop2
$ python -m pip install -e .
```

And test your `conan` installation:

```
$ conan
```

You should see the Conan commands help.

2.3 Update

If installed via `pip`, Conan 2.0 pre-release version can be easily updated:

```
$ pip install conan --pre --upgrade  # Might need sudo or --user
```

The default `<userhome>/.conan/settings.yml` file, containing the definition of compiler versions, etc., will be upgraded if Conan does not detect local changes, otherwise it will create a `settings.yml.new` with the new settings. If you want to regenerate the settings, you can remove the `settings.yml` file manually and it will be created with the new information the first time it is required.

The upgrade shouldn't affect the installed packages or cache information. If the cache becomes inconsistent somehow, you may want to remove its content by deleting it (`<userhome>/.conan`).

CHAPTER THREE

TUTORIAL

The purpose of this section is to guide you through the most important Conan features with practical examples. From using libraries already packaged by Conan, to how to package your libraries and store them in a remote server alongside all the precompiled binaries.

Important: This tutorial is part of the Conan 2.0 documentation. Conan 2.0 is still in alpha state. Some details, like the repositories and libraries used for the tutorial, will change as we update the current 1.X Conan packages to be compatible with Conan 2.0.

3.1 Consuming Packages

In this section, we first show how to get started with Conan: declaring your dependencies in a project, that can be libraries (like *zlib*, *openssl*, *boost*, etc.) or build tools (like *CMake*, *msys2*, *MinGW*, etc.), how to build for different configurations (like Release, Debug, Static, Shared, etc.). Also, how you can make a more advanced dependency declaration in your projects using a *conanfile.py* to make things like conditional requirements.

Then you will learn… (TODO: versioning part)

3.1.1 Getting started

This section shows how to build your projects using Conan to manage your dependencies. We will begin with basic example of a C project that uses CMake and depends on the **zlib** library. This project will use a *conanfile.txt* file to declare it's dependencies.

We will also cover how you can not only use 'regular' libraries with Conan but also manage tools you may need to use while building: like CMake, msys2, MinGW, etc.

Then, we will explain different Conan concepts like settings and options and how you can use them to build your projects for different configurations like Debug, Release, with static or shared libraries, etc.

Also, we will explain how to transition from the *conanfile.txt* file we used in the first example to a more powerful conanfile.py.

Build a simple CMake project using Conan

Let's get started with an example: We are going to create a string compressor application that uses one of the most popular C++ libraries: Zlib.

Important: In this example, we will retrieve the CMake Conan package from a Conan repository with packages compatible for Conan 2.0. To run this example succesfully you should add this remote to your Conan configuration (if did not already do it) doing: `conan remote add conanv2 https://conanv2beta.jfrog.io/artifactory/api/conan/conan --index 0`

We'll use CMake as build system in this case but keep in mind that Conan **works with any build system** and is not limited to using CMake. You can check more examples with other build systems in the *Read More section*.

Please, first clone the sources to recreate this project, you can find them in the examples2.0 repository in GitHub:

```
$ git clone https://github.com/conan-io/examples2.git
$ cd tutorial/consuming_packages/getting_started/simple_cmake_project
```

We start from a very simple C language project with this structure:

```
.
├── CMakeLists.txt
└── src
    └── main.c
```

This project contains a basic *CMakeLists.txt* including the **zlib** dependency and the source code for the string compressor program in *main.c*.

Let's have a look at the *main.c* file:

Listing 1: **main.c**

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include <zlib.h>

int main(void) {
    char buffer_in [256] = {"Conan is a MIT-licensed, Open Source package manager for␣
↪C and C++ development "
                            "for C and C++ development, allowing development teams to␣
↪easily and efficiently "
                            "manage their packages and dependencies across platforms␣
↪and build systems."};
    char buffer_out [256] = {0};

    z_stream defstream;
    defstream.zalloc = Z_NULL;
    defstream.zfree = Z_NULL;
    defstream.opaque = Z_NULL;
    defstream.avail_in = (uInt) strlen(buffer_in);
    defstream.next_in = (Bytef *) buffer_in;
    defstream.avail_out = (uInt) sizeof(buffer_out);
    defstream.next_out = (Bytef *) buffer_out;

    deflateInit(&defstream, Z_BEST_COMPRESSION);
    deflate(&defstream, Z_FINISH);
    deflateEnd(&defstream);

    printf("Uncompressed size is: %lu\n", strlen(buffer_in));
    printf("Compressed size is: %lu\n", strlen(buffer_out));

    printf("ZLIB VERSION: %s\n", zlibVersion());
```

(continues on next page)

(continued from previous page)

```
    return EXIT_SUCCESS;
}
```

Also, the contents of *CMakeLists.txt* are:

Listing 2: **CMakeLists.txt**

```
cmake_minimum_required(VERSION 3.15)
project(compressor C)

find_package(ZLIB REQUIRED)

add_executable(${PROJECT_NAME} src/main.c)
target_link_libraries(${PROJECT_NAME} ZLIB::ZLIB)
```

Our application relies on the **Zlib** library. Conan, by default, tries to install libraries from a remote server called ConanCenter. You can search there for libraries and also check the available versions. In our case, after checking the available versions for Zlib we choose to use the latest available version: **zlib/1.2.11**.

The easiest way to install the **Zlib** library and find it from our project with Conan is using a *conanfile.txt* file. Let's create one with the following content:

Listing 3: **conanfile.txt**

```
[requires]
zlib/1.2.11

[generators]
CMakeDeps
CMakeToolchain
```

As you can see we added two sections to this file with a syntax similar to an *INI* file.

- **[requires]** section is where we declare the libraries we want to use in the project, in this case **zlib/1.2.11**.
- **[generators]** section tells Conan to generate the files that the compilers or build systems will use to find the dependencies and build the project. In this case, as our project is based in *CMake*, we will use *CMakeDeps* to generate information about where the **Zlib** library files are installed and *CMakeToolchain* to pass build information to *CMake* using a *CMake* toolchain file.

Besides the *conanfile.txt*, we need a **Conan profile** to build our project. Conan profiles allows users to define a configuration set for things like compiler, build configuration, architecture, shared or static libraries, etc. Conan, by default, will not try to detect a profile automatically, so we need to create one. To let Conan try to guess the profile, based on the current operating system and installed tools, please run:

```
conan profile detect --force
```

This will detect the operating system, build architecture and compiler settings based on the environent. It will also set the build configuration as *Release* by default. The generated profile will be stored in the Conan home folder with name *default* and will be used by Conan in all commands by default unless other profile is specified via the command line. After executing the command you should see some output similar to this but for your configuration:

```
$ conan profile detect --force
CC and CXX: /usr/bin/gcc, /usr/bin/g++
Found gcc 10
gcc>=5, using the major as version
gcc C++ standard library: libstdc++11
```

(continues on next page)

(continued from previous page)

```
Detected profile:
[settings]
os=Linux
arch=x86_64
compiler=gcc
compiler.version=10
compiler.libcxx=libstdc++11
compiler.cppstd=gnu14
build_type=Release
[options]
[tool_requires]
[env]
...
```

We will use Conan to install **Zlib** and generate the files that CMake needs to find this library and build our project. We will generate those files in the folder *cmake-build-release* (Linux/macOS) or in the folder *build* (Windows). To do that, just run:

Listing 4: Windows

```
$ conan install . --output-folder=build --build=missing
```

Listing 5: Linux, macOS

```
$ conan install . --output-folder cmake-build-release --build=missing
```

You will get something similar to this as output of that command:

```
(Windows)
$ conan install . --output-folder=build --build=missing

(Linux, macOS)
$ conan install . --output-folder cmake-build-release --build=missing
...
-------- Computing dependency graph ----------
zlib/1.2.11: Not found in local cache, looking in remotes...
zlib/1.2.11: Checking remote: conanv2
zlib/1.2.11: Trying with 'conanv2'...
Downloading conanmanifest.txt
Downloading conanfile.py
Downloading conan_export.tgz
Decompressing conan_export.tgz
zlib/1.2.11: Downloaded recipe revision f1fadf0d3b196dc0332750354ad8ab7b
Graph root
    conanfile.txt: /home/conan/examples2/tutorial/consuming_packages/getting_started/
↪simple_cmake_project/conanfile.txt
Requirements
    zlib/1.2.11#f1fadf0d3b196dc0332750354ad8ab7b - Downloaded (conanv2)

-------- Computing necessary packages ----------
Requirements
    zlib/1.2.11
↪#f1fadf0d3b196dc0332750354ad8ab7b:cdc9a35e010a17fc90bb845108cf86cfcbce64bf
↪#dd7bf2a1ab4eb5d1943598c09b616121 - Download (conanv2)

-------- Installing packages ----------
```

(continues on next page)

(continued from previous page)

```

Installing (downloading, building) binaries...
zlib/1.2.11: Retrieving package cdc9a35e010a17fc90bb845108cf86cfcbce64bf from remote
↪'conanv2'
Downloading conanmanifest.txt
Downloading conaninfo.txt
Downloading conan_package.tgz
Decompressing conan_package.tgz
zlib/1.2.11: Package installed cdc9a35e010a17fc90bb845108cf86cfcbce64bf
zlib/1.2.11: Downloaded package revision dd7bf2a1ab4eb5d1943598c09b616121

-------- Finalizing install (deploy, generators) ----------
conanfile.txt: Generator 'CMakeToolchain' calling 'generate()'
conanfile.txt: Generator 'CMakeDeps' calling 'generate()'
conanfile.txt: Aggregating env generators
```

As you can see in the output, there are a couple of things that happened:

- Conan installed the *Zlib* library from the remote server we configured at the beginning of the tutorial. This server stores both the Conan recipes, that are the files that define how libraries must be built and binaries that can be reused so we don't have to build from sources everytime.
- Conan generated several files under the **cmake-build-release** folder. Those files were generated by both the `CMakeToolchain` and `CMakeDeps` generators we set in the **conanfile.txt**. `CMakeDeps` generates files so that CMake finds the Zlib library we have just download. On the other side `CMakeToolchain` generates a toolchain file for CMake so that we can transparently build our project with CMake using the same settings that we detected for our default profile.

Now we are ready to build and run our **compressor** app:

Listing 6: Windows

```
$ cd build
# assuming Visual Studio 15 2017 is your VS version and that it matches your default␣
↪profile
$ cmake .. -G "Visual Studio 15 2017" -DCMAKE_TOOLCHAIN_FILE=conan_toolchain.cmake
$ cmake --build . --config Release
...
[100%] Built target compressor
$ Release\compressor.exe
Uncompressed size is: 233
Compressed size is: 147
ZLIB VERSION: 1.2.11
```

Listing 7: Linux, macOS

```
$ cd cmake-build-release
$ cmake .. -DCMAKE_TOOLCHAIN_FILE=conan_toolchain.cmake
$ cmake --build .
...
[100%] Built target compressor
$ ./compressor
Uncompressed size is: 233
Compressed size is: 147
ZLIB VERSION: 1.2.11
```

Read more

- Getting started with Autotools
- Getting started with Meson
- ...

Using build tools as Conan packages

Important: In this example, we will retrieve the CMake Conan package from a Conan repository with packages compatible for Conan 2.0. To run this example succesfully you should add this remote to your Conan configuration (if did not already do it) doing: `conan remote add conanv2 https://conanv2beta.jfrog.io/artifactory/api/conan/conan --index 0`

In the previous example, we built our CMake project and used Conan to install and locate the **Zlib** library. Conan used the CMake version found in the system path to build this example. But, what happens if you don't have CMake installed in your build environment or want to build your project with a specific CMake version different from the one you have already installed system-wide? In this case, you can declare this dependency in Conan using a type of requirement named `tool_requires`. Let's see an example on how to add a `tool_requires` to our project, and using a different CMake version to build it.

Please, first clone the sources to recreate this project. You can find them in the examples2.0 repository in GitHub:

```
$ git clone https://github.com/conan-io/examples2.git
$ cd tutorial/consuming_packages/getting_started/tool_requires
```

The structure of the project is the same as the one of the previous example:

```
.
├── conanfile.txt
├── CMakeLists.txt
└── src
    └── main.c
```

The main difference is the addition of the **[tool_requires]** section in the **conanfile.txt** file. In this section, we declare that we want to build our application using CMake **v3.19.8**.

Listing 8: **conanfile.txt**

```
[requires]
zlib/1.2.11

[tool_requires]
cmake/3.19.8

[generators]
CMakeDeps
CMakeToolchain
```

We also added a message to the *CMakeLists.txt* to output the CMake version:

Listing 9: **CMakeLists.txt**

```
cmake_minimum_required(VERSION 3.15)
project(compressor C)
```

(continues on next page)

(continued from previous page)

```

find_package(ZLIB REQUIRED)

message("Building with CMake version: ${CMAKE_VERSION}")

add_executable(${PROJECT_NAME} src/main.c)
target_link_libraries(${PROJECT_NAME} ZLIB::ZLIB)
```

Now, as in the previous example, we will use Conan to install **Zlib** and **CMake 3.19.8** and generate the files to find both of them. We will generate those files in the folder *cmake-build-release* (Linux/macOS) or in the folder *build* (Windows). To do that, just run:

Listing 10: Windows

```
$ conan install . --output-folder=build --build=missing
```

Listing 11: Linux, macOS

```
$ conan install . --output-folder cmake-build-release --build=missing
```

You can check the output:

```
-------- Computing dependency graph ----------
cmake/3.19.8: Not found in local cache, looking in remotes...
cmake/3.19.8: Checking remote: conanv2
cmake/3.19.8: Trying with 'conanv2'...
Downloading conanmanifest.txt
Downloading conanfile.py
cmake/3.19.8: Downloaded recipe revision 3e3d8f3a848b2a60afafbe7a0955085a
Graph root
    conanfile.txt: /Users/carlosz/Documents/developer/conan/examples2/tutorial/
↪consuming_packages/getting_started/tool_requires/conanfile.txt
Requirements
    zlib/1.2.11#f1fadf0d3b196dc0332750354ad8ab7b - Cache
Build requirements
    cmake/3.19.8#3e3d8f3a848b2a60afafbe7a0955085a - Downloaded (conanv2)

-------- Computing necessary packages ----------
Requirements
    zlib/1.2.11
↪#f1fadf0d3b196dc0332750354ad8ab7b:2a823fda5c9d8b4f682cb27c30caf4124c5726c8
↪#48bc7191ec1ee467f1e951033d7d41b2 - Cache
Build requirements
    cmake/3.19.8
↪#3e3d8f3a848b2a60afafbe7a0955085a:f2f48d9745706caf77ea883a5855538256e7f2d4
↪#6c519070f013da19afd56b52c465b596 - Download (conanv2)

-------- Installing packages ----------

Installing (downloading, building) binaries...
cmake/3.19.8: Retrieving package f2f48d9745706caf77ea883a5855538256e7f2d4 from remote
↪'conanv2'
Downloading conanmanifest.txt
Downloading conaninfo.txt
Downloading conan_package.tgz
Decompressing conan_package.tgz
```

(continues on next page)

(continued from previous page)

```
cmake/3.19.8: Package installed f2f48d9745706caf77ea883a5855538256e7f2d4
cmake/3.19.8: Downloaded package revision 6c519070f013da19afd56b52c465b596
zlib/1.2.11: Already installed!

-------- Finalizing install (deploy, generators) ----------
conanfile.txt: Generator 'CMakeToolchain' calling 'generate()'
conanfile.txt: Generator 'CMakeDeps' calling 'generate()'
conanfile.txt: Aggregating env generators
```

Now, if you check the folder you will see that Conan generated a new file called `conanbuild.sh/bat`. This is the result of automatically invoking a `VirtualBuildEnv` generator when we declared the `tool_requires` in the **conanfile.txt**. This file sets some environment variables like a new `PATH` that we can use to inject to our environment the location of CMake v3.19.8.

Activate the virtual environment, and run `cmake --version` to check that you have the installed the new CMake version in the path.

Listing 12: Windows

```
$ cd build
$ conanbuild.bat
```

Listing 13: Linux, macOS

```
$ cd cmake-build-release
$ source conanbuild.sh
Capturing current environment in deactivate_conanbuildenv-release-x86_64.sh
Configuring environment variables
```

Run `cmake` and check the version:

```
$ cmake --version
cmake version 3.19.8
...
```

As you can see, after activating the environment, the CMake v3.19.8 binary folder was added to the path and is the current active version now. Now you can build your project as you prevoiusly did, but this time Conan will use CMake 3.19.8 to build it:

Listing 14: Windows

```
# assuming Visual Studio 15 2017 is your VS version and that it matches your default␣
↪profile
$ cmake .. -G "Visual Studio 15 2017" -DCMAKE_TOOLCHAIN_FILE=conan_toolchain.cmake
$ cmake --build . --config Release
...
Building with CMake version: 3.19.8
...
[100%] Built target compressor
$ Release\compressor.exe
Uncompressed size is: 233
Compressed size is: 147
ZLIB VERSION: 1.2.11
```

Listing 15: Linux, macOS

```
$ cmake .. -DCMAKE_TOOLCHAIN_FILE=conan_toolchain.cmake
$ cmake --build .
...
Building with CMake version: 3.19.8
...
[100%] Built target compressor
$ ./compressor
Uncompressed size is: 233
Compressed size is: 147
ZLIB VERSION: 1.2.11
```

Note that when we activated the environment, a new file named `deactivate_conanbuild.sh/bat` was created in the same folder. If you source this file you can restore the environment as it was before.

Listing 16: Windows

```
$ deactivate_conanbuild.bat
```

Listing 17: Linux, macOS

```
$ source deactivate_conanbuild.sh
Restoring environment
```

Run `cmake` and check the version, it will be the version that was installed previous to the environment activation:

```
$ cmake --version
cmake version 3.22.0
...
```

Read more

- Using MinGW as tool_requires
- Using tool_requires in profiles
- Using conf to set a toolchain from a tool requires
- Creating recipes for tool_requires: packaging build tools

Building for multiple configurations: Release, Debug, Static and Shared

Understanding the power of using conanfile.py vs conanfile.txt

3.2 Creating Packages

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

3.2.1 Getting started

This section introduces how to create your own Conan packages, explain *conanfile.py* recipes and the commands to build packages from sources in your computer.

Important: This is a **tutorial** section. You are encouraged to execute these commands. For this concrete example, you will need **CMake** installed in your path. It is not strictly required by Conan to create packages, you can use other build systems (as VS, Meson, Autotools and even your own) to do that, without any dependency to CMake.

Using the **conan new** command will create a "Hello World" C++ library example project for us:

```
$ mkdir hellopkg && cd hellopkg
$ conan new hello/0.1 --template=cmake_lib
File saved: CMakeLists.txt
File saved: conanfile.py
File saved: src/hello.cpp
File saved: src/hello.h
File saved: test_package/CMakeLists.txt
File saved: test_package/conanfile.py
File saved: test_package/src/example.cpp
```

The generated files are:

- **conanfile.py**: On the root folder, there is a *conanfile.py* which is the main recipe file, responsible for defining how the package is built and consumed.
- **CMakeLists.txt**: A simple generic *CMakeLists.txt*, with nothing specific about Conan in it.
- **src** folder: the *src* folder that contains the simple C++ "hello" library.
- (optional) **test_package** folder: contains an *example* application that will require and link with the created package. It is not mandatory, but it is useful to check that our package is correctly created.

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

```
from conans import ConanFile
from conan.tools.cmake import CMakeToolchain, CMake
from conan.tools.layout import cmake_layout


class HelloConan(ConanFile):
    name = "hello"
    version = "0.1"

    # Optional metadata
    license = "<Put the package license here>"
    author = "<Put your name here> <And your email here>"
    url = "<Package recipe repository url here, for issues about the package>"
    description = "<Description of Hello here>"
    topics = ("<Put some tag here>", "<here>", "<and here>")

    # Binary configuration
    settings = "os", "compiler", "build_type", "arch"
    options = {"shared": [True, False], "fPIC": [True, False]}
    default_options = {"shared": False, "fPIC": True}

    # Sources are located in the same place as this recipe, copy them to the recipe
    exports_sources = "CMakeLists.txt", "src/*"

    def config_options(self):
        if self.settings.os == "Windows":
            del self.options.fPIC

    def layout(self):
        cmake_layout(self)

```

(continues on next page)

(continued from previous page)

```
    def generate(self):
        tc = CMakeToolchain(self)
        tc.generate()

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

    def package(self):
        cmake = CMake(self)
        cmake.install()

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

Let's explain this recipe a little bit:

- The binary configuration is composed by `settings` and `options`. When something changes in the configuration, the resulting binary built and packaged will be different:
  - `settings` are project wide configuration that cannot be defaulted in recipes, like the OS or the architecture.
  - `options` are package specific configuration and can be defaulted in recipes, in this case we have the option of creating the package as a shared or static library, being static the default.
- The `exports_sources` attribute defines which sources are exported together with the recipe, these sources become part of the package recipe (there are other mechanisms that don't do this, will be explained later.
- The `config_options()` method (together with `configure()` one) allows to fine tune the binary configuration model, for example, in Windows there is no `fPIC` option, so it can be removed.
- The `generate()` method prepares the build of the package from source. In this case, it could be simplified to an attribute `generators = "CMakeToolchain"`, but it is left to show this important method. In this case, the execution of `CMakeToolchain` `generate()` method will create a *conan_toolchain.cmake* file that translates the Conan `settings` and `options` to CMake syntax.
- The `build()` method uses the `CMake` wrapper to call CMake commands, it is a thin layer that will manage to pass in this case the `-DCMAKE_TOOLCHAIN_FILE=conan_toolchain.cmake` argument. It will configure the project and build it from source.
- The `package()` method copies artifacts (headers, libs) from the build folder to the final package folder. It can be done with bare "copy" commands, but in this case it is leveraging the already existing CMake install functionality (if the CMakeLists.txt didn't implement it, it is easy to write `self.copy()` commands in this `package()` method.
- Finally, the `package_info()` method defines that consumers must link with a "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 (as `CMakeDeps`) to be used by consumers. Although this method implies some potential duplication with the build system output (CMake could generate xxx-config.cmake files), it is important to define this, as Conan packages can be consumed by any other build system, not only CMake.

The contents of the `test_package` folder is not critical now for understanding how packages are created. The important bits are:

- `test_package` folder is different from unit or integration tests. These tests are "package" tests, and validate that the package is properly created, and that the package consumers will be able to link against it and reuse it.

- It is a small Conan project itself, it contains its own `conanfile.py`, and its source code including build scripts, that depends on the package being created, and builds and execute a small application that requires the library in the package.
- It doesn't belong to the package. It only exist in the source repository, not in the package.

Let's build the package from sources with the current default configuration, and then let the `test_package` folder test the package:

```
$ conan create . demo/testing
...
hello/0.1: Hello World Release!
  hello/0.1: _M_X64 defined
  ...
```

If "Hello world Release!" is displayed, it worked. This is what has happened:

- The *conanfile.py* together with the contents of the *src* folder have been copied (exported, in Conan terms) to the local Conan cache.
- A new build from source for the `hello/0.1@demo/testing` package starts, calling the `generate()`, `build()` and `package()` methods. This creates the binary package in the Conan cache.
- Moves to the *test_package* folder and executes a `conan install` + `conan build` + `test()` method, to check if the package was correctly created.

We can now validate that the recipe and the package binary are in the cache:

```
$ conan list recipes hello
Local Cache:
  hello
    hello/0.1@demo/testing#afa4685e137e7d13f2b9845987c5af77

$ conan list package-ids hello/0.1@demo/testing#afa4685e137e7d13f2b9845987c5af77
Local Cache:
  hello/0.1@demo/testing
↪#afa4685e137e7d13f2b9845987c5af77:e360b62ce00057522e221cfe56714705a46e20e2
    settings:
      arch=x86_64
      build_type=Release
      compiler=apple-clang
      compiler.libcxx=libc++
      compiler.version=12.0
      os=Macos
    options:
      fPIC=True
      shared=False
```

The **`conan create`** command receives the same parameters as **`conan install`**, so you can pass to it the same settings and options. If we execute the following lines, we will create new package binaries for those configurations:

```
$ conan create . demo/testing -s build_type=Debug
...
hello/0.1: Hello World Debug!

$ conan create . demo/testing -o hello:shared=True
...
hello/0.1: Hello World Release!
```

These new package binaries will be also stored in the Conan cache, ready to be used by any project in this computer, we can see them with:

```
$ conan list package-ids hello/0.1@demo/testing#afa4685e137e7d13f2b9845987c5af77
Local Cache:
  hello/0.1@demo/testing
↪#afa4685e137e7d13f2b9845987c5af77:842490321f80b0a9e1ba253d04972a72b836aa28
    settings:
      arch=x86_64
      build_type=Release
      compiler=apple-clang
      compiler.libcxx=libc++
      compiler.version=12.0
      os=Macos
    options:
      fPIC=True
      shared=True
  hello/0.1@demo/testing
↪#afa4685e137e7d13f2b9845987c5af77:a5c01fc21d2db712d56189dff69fc10f12b22375
    settings:
      arch=x86_64
      build_type=Debug
      compiler=apple-clang
      compiler.libcxx=libc++
      compiler.version=12.0
      os=Macos
    options:
      fPIC=True
      shared=False
  hello/0.1@demo/testing
↪#afa4685e137e7d13f2b9845987c5af77:e360b62ce00057522e221cfe56714705a46e20e2
    settings:
      arch=x86_64
      build_type=Release
      compiler=apple-clang
      compiler.libcxx=libc++
      compiler.version=12.0
      os=Macos
    options:
      fPIC=True
      shared=False
```

Any doubts? Please check out our *FAQ section* or open a Github issue

CHAPTER FOUR

INTEGRATIONS

CHAPTER

FIVE

EXAMPLES

CHAPTER

SIX

REFERENCE

6.1 Conan commands

6.1.1 conan search

Search existing recipes in remotes. This command is equivalent to `conan list recipes <query> -r=*`, and is provided for simpler UX.

```
conan search -h
usage: conan search [-h] [-f {cli,json}] [-r REMOTE] query

Searches for package recipes in a remote or remotes

positional arguments:
query                 Search query to find package recipe reference, e.g., 'boost',
↪'lib*'

optional arguments:
-h, --help            show this help message and exit
-f {cli,json}, --format {cli,json}
                        Select the output format: cli, json. 'cli' is the default␣
↪output.
-r REMOTE, --remote REMOTE
                        Remote names. Accepts wildcards. If not specified it searches␣
↪in all remotes
```

```
$ conan search zlib
conancenter:
zlib
    zlib/1.2.11
    zlib/1.2.8

$ conan search zlib -r=conancenter
conancenter:
zlib
    zlib/1.2.11
    zlib/1.2.8

$ conan search zlib/1.2.1* -r=conancenter
conancenter:
zlib
    zlib/1.2.11

$ conan search zlib/1.2.1* -r=conancenter --format=json
```

(continues on next page)

(continued from previous page)

```
[
    {
        "remote": "conancenter",
        "error": null,
        "results": [
            {
                "name": "zlib",
                "id": "zlib/1.2.11"
            }
        ]
    }
]
```

6.1.2 conan list

conan list recipes

```
$ conan list recipes zlib -r=conancenter
conancenter:
zlib
    zlib/1.2.11
    zlib/1.2.8

$ conan list recipes zlib/1.2.1* -r=conancenter
conancenter:
zlib
    zlib/1.2.11

$ conan list recipes zlib/1.2.1* -r=conancenter --format=json
[
    {
        "remote": "conancenter",
        "error": null,
        "results": [
            {
                "name": "zlib",
                "id": "zlib/1.2.11"
            }
        ]
    }
]
```

conan list package-ids

```
$ conan list package-ids zlib/1.2.11 -r=conancenter
...
zlib/1.2.11:1513b3452ef7e2a2dd5f931247c5e02edeb98cc9
    settings:
    os=Macos
    arch=x86_64
    compiler=apple-clang
    build_type=Debug
    compiler.version=10.0
    options:
    shared=False
```

(continues on next page)

(continued from previous page)

```
    fPIC=True
zlib/1.2.11:963bb116781855de98dbb23aaac41621e5d312d8
    settings:
    os=Windows
    compiler.runtime=MTd
    arch=x86_64
    compiler=Visual Studio
    build_type=Debug
    compiler.version=15
    options:
    shared=False
zlib/1.2.11:bf6871a88a66b609883bce5de4dd61adb1e033a7
    settings:
    os=Linux
    arch=x86_64
    compiler=gcc
    build_type=Debug
    compiler.version=5
    options:
    shared=True
...
```

conan list recipe-revisions

```
$ conan list recipe-revisions zlib/1.2.11 -r=conancenter
conancenter:
...
  zlib/1.2.11#b3eaf63da20a8606f3d84602c2cfa854 (2021-08-27T20:02:46Z)
  zlib/1.2.11#08c5163c8e302d1482d8fa2be93736af (2021-05-05T16:17:39Z)
  zlib/1.2.11#b291478a29f383b998e1633bee1c0536 (2021-03-25T10:03:21Z)
  zlib/1.2.11#514b772abf9c36ad9be48b84cfc6fdc2 (2021-02-19T14:33:26Z)
```

conan list package-revisions

```
$conan list package-revisions zlib/1.2.11
↪#b3eaf63da20a8606f3d84602c2cfa854:963bb116781855de98dbb23aaac41621e5d312d8 -
↪r=conancenter
conancenter:
  zlib/1.2.11
↪#b3eaf63da20a8606f3d84602c2cfa854:963bb116781855de98dbb23aaac41621e5d312d8
↪#dd44f4a86108e836f0c2d35af89cd8cd (2021-08-27T20:12:00Z)
```

6.1.3 Creator commands

6.2 Python API

CHAPTER SEVEN

FAQ

See also:

There is a great community behind Conan with users helping each other in Cpplang Slack. Please join us in the `#conan` channel!