



Conan Documentation

Release 2.0.0-alpha

The Conan team

Jan 13, 2022

CONTENTS

- 1 Introduction** **1**
- 2 Install** **3**
 - 2.1 Install with pip (recommended) 3
 - 2.2 Install from source 4
 - 2.3 Update 4
- 3 Tutorial** **5**
 - 3.1 Creating Packages 5
- 4 Integrations** **11**
- 5 Examples** **13**
- 6 Reference** **15**
 - 6.1 Conan commands 15
 - 6.2 Python API 17
- 7 FAQ** **19**

INTRODUCTION

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 Conan using `pip`, you need a Python `>= 3.5` 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 `<= 3.4` support has been dropped. Conan will not work with those python versions.

Install Conan:

```
$ pip install conan==2.0.0a1
```

Important: Please READ carefully

- Make sure that your **pip** installation matches your **Python (`>= 3.5`)** 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 can be easily updated:

```
$ pip install conan --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`).

TUTORIAL

3.1 Creating Packages

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

3.1.1 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
```

(continues on next page)

(continued from previous page)

```

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)

    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#a4a4685e137e7d13f2b9845987c5af77

$ conan list package-ids hello/0.1@demo/testing#a4a4685e137e7d13f2b9845987c5af77
```

(continues on next page)

(continued from previous page)

```

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#a4685e137e7d13f2b9845987c5af77
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

```

(continues on next page)

(continued from previous page)

```
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](#)

INTEGRATIONS

EXAMPLES

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

See also:

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