Conan Documentation

Release 2.0.1

The Conan team

Mar 06, 2023

CONTENTS

Welcome! This is the user documentation for Conan, an open source, decentralized C/C++ package manager that works in all platforms and with all build systems and compilers. Other relevant resources:

- Conan home page. Entry point to the project, with links to docs, blog, social, downloads, release mailing list, etc.
- Github project and issue tracker. The main support channel, file issues here for questions, bug reports and feature requests.

Table of contents:

CHAPTER

ONE

INTRODUCTION

Conan is a dependency and package manager for C and C++ languages. It is free and open-source, works in all platforms (Windows, Linux, OSX, FreeBSD, Solaris, etc.), and can be used to develop for all targets including embedded, mobile (iOS, Android), and bare metal. It also integrates with all build systems like CMake, Visual Studio (MSBuild), Makefiles, SCons, etc., including proprietary ones.

It is specifically designed and optimized for accelerating the development and Continuous Integration of C and C++ projects. With full binary management, it can create and reuse any number of different binaries (for different configurations like architectures, compiler versions, etc.) for any number of different versions of a package, using exactly the same process in all platforms. As it is decentralized, it is easy to run your own server to host your own packages and binaries privately, without needing to share them. The free JFrog Artifactory Community Edition (CE) is the recommended Conan server to host your own packages privately under your control.

Conan is mature and stable, with a strong commitment to forward compatibility (non-breaking policy), and has a complete team dedicated full time to its improvement and support. It is backed and used by a great community, from open source contributors and package creators in ConanCenter to thousands of teams and companies using it.

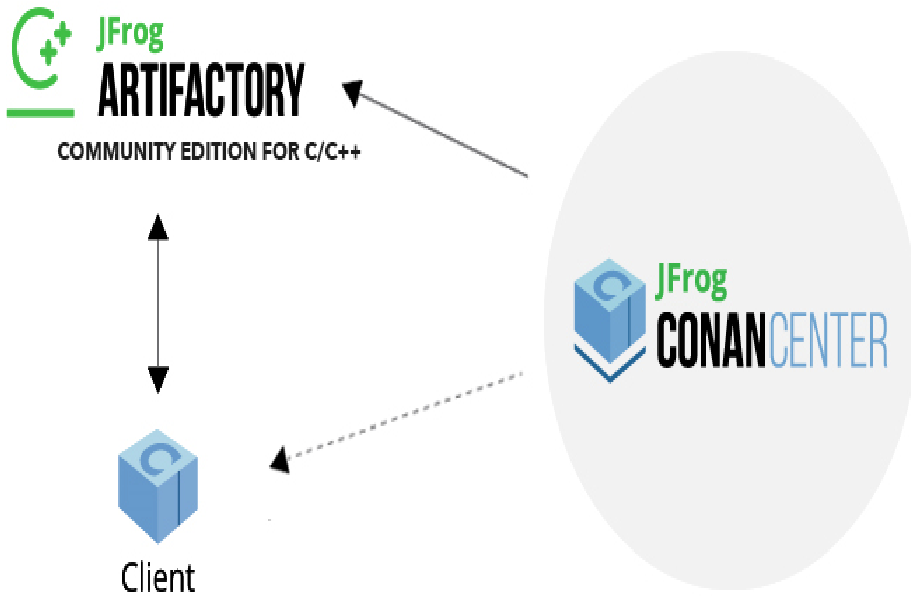
1.1 Open Source

Conan is Free and Open Source, with a permissive MIT license. Check out the source code and issue tracking (for questions and support, reporting bugs and suggesting feature requests and improvements) at https://github.com/conan-io/conan

1.2 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.

At a high level, the servers are just storing packages. 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.



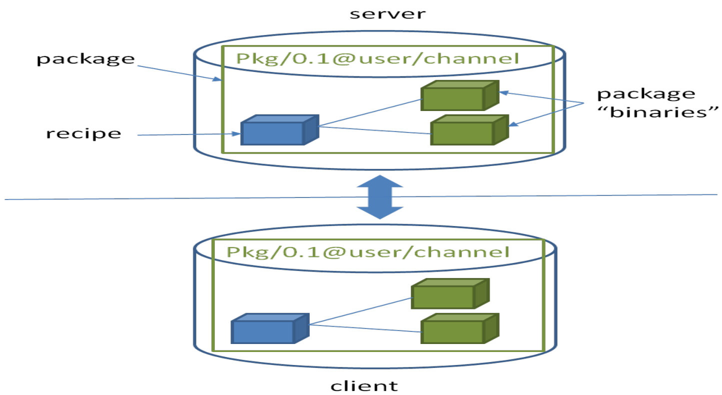
The different applications in the image above are:

- 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 as long as no new packages are needed from remote servers.
- JFrog Artifactory Community Edition (CE) is the recommended Conan server to host your own packages privately under your control. It is a free community edition of JFrog Artifactory for Conan packages, including a WebUI, multiple auth protocols (LDAP), Virtual and Remote repositories to create advanced topologies, a Rest API, and generic repositories to host any artifact.
- The conan_server is a small server distributed together with the Conan client. It is a simple open-source implementation and provides basic functionality, but no WebUI or other advanced features.
- ConanCenter is a central public repository where the community contributes packages for popular open-source libraries like Boost, Zlib, OpenSSL, Poco, etc.

1.3 Binary management

One of the most powerful features of Conan is that it can create and manage pre-compiled binaries for any possible platform and configuration. By using pre-compiled binaries and avoiding repeated builds from source, it saves significant time for developers and Continuous Integration servers, while also improving the reproducibility and traceability of artifacts.

A package is defined by a "conanfile.py". This is a file that defines the package's dependencies, sources, how to build the binaries from sources, etc. One package "conanfile.py" recipe can generate any arbitrary number of binaries, one for each different platform and configuration: operating system, architecture, compiler, build type, etc. These binaries can be created and uploaded to a server with the same commands in all platforms, having a single source of truth for all packages and not requiring a different solution for every different operating system.



Installation of packages from servers is also very efficient. Only the necessary binaries for the current platform and configuration are downloaded, not all of them. If the compatible binary is not available, the package can be built from sources in the client too.

1.4 All platforms, all build systems and compilers

Conan works 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. It can target any existing platform: ranging from bare metal to desktop, mobile, embedded, servers, and cross-building.

Conan works with any build system too. There are built-in integrations to support the most popular ones like CMake, Visual Studio (MSBuild), Autotools and Makefiles, Meson, SCons, etc., but it is not a requirement to use any of them. It is not even necessary that all packages use the same build system: each package can use their own build system, and depend on other packages using different build systems. It is also possible to integrate with any build system, including proprietary ones.

Likewise, Conan can manage any compiler and any version. There are default definitions for the most popular ones: gcc, cl.exe, clang, apple-clang, intel, with different configurations of versions, runtimes, C++ standard library, etc. This model is also extensible to any custom configuration.

1.5 Stable

From Conan 2.0 and onwards, there is a commitment to stability, with the goal of not breaking user space while evolving the tool and the platform. This means:

- Moving forward to following minor versions 2.1, 2.2, . . . , 2.X should never break existing recipes, packages or command line flows
- If something is breaking, it will be considered a regression and reverted.
- Bug fixes will not be considered breaking, recipes and packages relying on the incorrect behavior of such bugs will be considered already broken.
- Only documented features in https://docs.conan.io are considered part of the public interface of Conan. Private implementation details, and everything not included in the documentation is subject to change.

- The compatibility is always considered forward. New APIs, tools, methods, helpers can be added in following 2.X versions. Recipes and packages created with these features will be backwards incompatible with earlier Conan versions.
- Only the latest released patch (major.minor.patch) of every minor version is supported and stable.

There are some things that are not included in this commitment:

- Public repositories, like **ConanCenter**, 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. It is recommended to use your own private repository to store your own copy of the packages for production, or as a secondary alternative, to use some locking mechanism to avoid possible disruption from packages in ConanCenter that are updated and require latest Conan version.
- Configuration and automatic tools detection, like the detection of the default profile (`conan profile detect`) can and will change at any time. Users are encouraged to define their configurations in their own profiles files for repeatability. New versions of Conan might detect different default profiles.
- Builtin default implementation of extension points as plugins or hooks can also change with every release. Users can provide their own ones for stability.
- Output of packages templates with `conan new` can update at any time to use latest features.
- The output streams stdout, stderr, i.e. the terminal output can change at any time. Do not parse the terminal output for automation.
- Anything that is explicitily labeled as `experimental` or `preview` in the documentation, or in the Conan cli output.
- Anything that is labeled as `deprecated` in the documentation should not get new usages, as it will not get new fixes and it will be removed in the next major version.
- Other tools and repositories outside of the Conan client

Conan needs Python>=3.6 to run. Conan will deprecate support for Python versions 1 year after those versions have been declared End Of Life (EOL).

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.

1.6 Community

Conan is being used in production by thousands of companies like TomTom, Audi, RTI, Continental, Plex, Electrolux and Mercedes-Benz and many thousands of developers around the world.

But an essential part of Conan is that many of those users will contribute back, creating an amazing and helpful community:

- The https://github.com/conan-io/conan project has around 6.5K stars in Github and counts with contributions from more than 300 different users (this is just the client tool).
- Many other users contribute recipes for ConanCenter via the https://github.com/conan-io/conan-center-index repo, creating packages for popular Open Source libraries, contributing many thousands of Pull Requests per year.
- More than two thousands Conan users hang around the CppLang Slack #conan channel, and help responding to questions, discussing problems and approaches, making it one of the most active channels in the whole CppLang slack.
- There is a Conan channel in #include<cpp> discord.

1.7 Navigating the documentation

This documentation has very different sections:

- The **tutorial** is an actual hands-on tutorial, with examples and real code, intended to be played sequentially from beginning to end, running the exercises in your own computer. There is a "narrative" to this section and the exercises might depend on some previous explanations and code - building on the previous example. This is the recommended approach for learning Conan.
- The **examples** also contain hands-on, fully operational examples with code, aimed to explain some very specific feature, tool or behavior. They do not have a conducting thread, they should be navigated by topic.
- The **reference** is the source of truth for the interfaces of every public command, class, method, helper, API and configuration file that can be used. It is not designed to be read fully, but to check for individual items when necessary.
- The **knowledge** base contains things like the FAQ, a very important section about general guidelines, good practices and bad practices, videos from conference talks, etc.

Features in this documentation might be labeled as:

- **experimental**: This feature is released and can be used, but it is under active development and the interfaces, APIs or behavior might change as a result of evolution, and this will not be considered breaking. If you are interested in these features you are encouraged to try them and give feedback, because that is exactly what allows to stabilize them.
- **preview**: When a feature is released in preview mode, this means it aims to be as final and stable as possible. Users are encouraged to use them, and the maintainers team will try not to break them unless necessary. But if necessary, they might change and break.
- **deprecated**: This feature should no longer be used, and it will be fully removed in next major release. Other alternatives or approaches should be used instead of it, and if using it, migrating to the other alternatives should be done as soon as possible. They will not be maintained or get fixes.

Everything else that is not labeled should be considered stable, and won't be broken, unless something that is declared a bugfix.

Have any questions? Please check out our *FAQ section* or .

CHAPTER

TWO

WHAT'S NEW IN CONAN 2.0

Conan 2.0 comes with many exciting improvements based on the lessons learned in the last years with Conan 1.X. Also, a lot of effort has been made to backport necessary things to Conan 1.X to make the upgrade easier: Recipes using latest 1.X integrations will be compatible with Conan 2.0, and binaries for both versions will not collide and be able to live in the same server repositories.

2.1 Conan 2.0 migration guide

If you are using Conan 1.X, please read the Conan 2.0 Migration guide , to start preparing your package recipes to 2.0 and be aware of some changes while you still work in Conan 1.X. That guide summarizes the above mentioned backports to make the upgrade easier.

2.2 New graph model

Conan 2.0 defines new requirement traits (headers, libs, build, run, test, package_id_mode, options, transitive_headers, transitive_libs) and package types (static, shared, application, header-only) to better represent the relations that happen with C and C++ binaries, like executables or shared libraries linking static libraries or shared libraries.

Read more:

- https://www.youtube.com/watch?v=kKGglzm5ous
- https://github.com/conan-io/tribe/blob/main/design/026-requirements_traits.md
- https://github.com/conan-io/tribe/blob/main/design/027-package_types.md

2.3 New public Python API

A new modular Python API is made available, public and documented. This is a real API, with building blocks that are already used to build the Conan built-in commands, but that will allow further extensions. There are subapis for different functional groups, like `api.list`, `api.search`, `api.remove`, `api.profile`, `api.graph`, `api.upload`, `api.remotes`, etc. that will allow to implement advanced user flows, functionality and automation.

Read more:

- *Python API reference*

2.4 New build system integrations

Introduced in latest Conan 1.X, Conan 2.0 will use modern build system integrations like `CMakeDeps` and `CMakeToolchain` that are fully transparent CMake integration (i.e. the consuming `CMakeLists.txt` doesn't need to be aware at all about Conan). These integrations can also achieve a better IDE integration, for example via CMakePresets.json.

Read more:

- *Tools reference*

2.5 New custom user commands

Conan 2.0 allows extending Conan with custom user commands, written in python that can be called as `conan xxxx`. These commands can be shared and installed with `conan config install`, and have layers of commands and subcommands. The custom user comands use the new 2.0 public Python API to implement their functionality.

2.6 New CLI

Conan 2.0 has redesigned the CLI for better consistency, removing ambiguities, and improving the user experience. The new CLI also sends all the information, warning, and error messages to stderr, while keeping the final result in stdout, allowing multiple output formats like `--format=html` or `--format=json` and using redirects to create files `--format=json > myfile.json`. The information provided by the CLI will be more structured and thorough so that it can be used more easily for automation, especially in CI/CD systems.

Read more:

- *Commands reference*

2.7 New deployers

Conan 2.0 implements "deployers", which can be called in the command line as `conan install .... --deploy=mydeploy`, typically to perform copy operations from the Conan cache to user folders. Such deployers can be built-in ("full_deploy" and "direct_deploy" are provided so far), or user-defined, which can be shared and managed with `conan config install`. Deployers run before generators, and they can change the target folders. For example, if the `--deploy=full_deploy` deployer runs before `CMakeDeps`, the files generated by `CMakeDeps` will point to the local copy in the user folder done by the `full_deploy` deployer, and not to the Conan cache.

Deployers can be multi-configuration. Running `conan install . --deploy=full_deploy` repeatedly for different profiles, can achieve a fully self-contained project, including all the artifacts, binaries, and build files that is completely independent of Conan and no longer require Conan at all to build.

2.8 New package_id

Conan 2.0 defines a new, dynamic `package_id` that is a great improvement over the limitations of Conan 1.X. This `package_id` will take into account the package types and types of requirements to implement a more meaningful strategy, depending on the scenario. For example, it is well known that when an application `myapp` is linking a static library `mylib`, any change in the binary of the static library `mylib` requires re-building the application `myapp`. So Conan will default to a mode like `full_mode` that will generate a new `myapp` `package_id`, for every change in the mylib recipe or binary. While a dependency between a static library mylib_a that is used by‘‘mylib_b‘‘ in general does not imply that a change in mylib_b always needs a rebuild of mylib_a, and that relationship can default to a minor_mode mode. In Conan 2.0, the one doing modifications to mylib_a can better express whether the consumer mylib_b needs to rebuild or not, based on the version bump (patch version bump will not trigger a rebuild while a minor version bump will trigger it)

Furthermore the default versioning scheme in Conan has been generalized to any number of digits and letters, as opposed to the official "semver" that uses just 3 fields.

2.9 compatibility.py

Conan 2.0 features a new extension mechanism to define binary compatibility at a global level. A compatibility.py file in the Conan cache will be used to define which fallbacks of binaries should be used in case there is some missing binary for a given package. Conan will provide a default one to account for cppstd compatibility, and executables compatibility, but this extension is fully configurable by the user (and can also be shared and managed with conan config install)

2.10 New lockfiles

Lockfiles in Conan 2.0 have been greatly simplified and made way more flexible. Lockfiles are now modeled as lists of sorted references, which allow one single lockfile being used for multiple configurations, merging lockfiles, applying partially defined lockfiles, being strict or non-strict, adding user defined constraints to lockfiles, and much more.

Read more:

- *Tutorial introduction to lockfiles*
- https://github.com/conan-io/tribe/blob/main/design/034-new_lockfiles.md
- *Tutorial about versioning and lockfiles*

2.11 New configuration and environment management

The new configuration system called [conf] in profiles and command line, and introduced experimentally in Conan 1.X, is now the major mechanism to configure and control Conan behavior. The idea is that the configuration system is used to transmit information from Conan (a Conan profile) to Conan (A Conan recipe, or a Conan build system integration like CMakeToolchain). This new configuration system can define strings, boolean, lists, being cleaner, more structured and powerful than using environment variables. A better, more explicit environment management, also introduced in Conan 1.X is now the way to pass information from Conan (profiles) to tools (like compilers, build systems).

Read more:

- *Reference of enviroment tools*

2.12 Multi-revision cache

The Conan cache has been completely redesigned to allow storing more than one revision at a time. It has also shortened the paths, using hashes, removing the need to use short_paths in Windows. Note that the cache is still not concurrent, so parallel jobs or tasks should use independent caches.

2.13 New extensions plugins

Several extension points, named "plugins" have been added, to provide advanced and typically orthogonal functionality to what the Conan recipes implement. These plugins can be shared, managed and installed via `conan config install`

2.13.1 Profile checker

A new `profile.py` extension point is provided that can be used to perform operations on the profile after it has been processed. A default implementation that checks that the given compiler version is capable of supporting the given compiler `cppstd` is provided, but this is fully customizable by the user.

2.13.2 Command wrapper

A new `cmd_wrapper.py` extension provides a way to wrap any `conanfile.py` command (i.e., anything that runs inside `self.run()` in a recipe), in a new command. This functionality can be useful for wrapping build commands in build optimization tools as IncrediBuild or compile caches.

2.13.3 Package signing

A new `sign.py` extension has been added to implement signing and verifying of packages. As the awareness about the importance of software supply chain security grows, it is becoming more important the capability of being able to sign and verify software packages. This extension point will soon get a plugin implementation based on Sigstore.

2.14 Package immutability optimizations

The thorough use of `revisions` (already introduced in Conan 1.X as opt-in in https://docs.conan.io/en/latest/versioning/revisions.html) in Conan 2.0, together with the declaration of artifacts **immutability** allows for improved processes, downloading, installing and updated dependencies as well as uploading dependencies.

The `revisions` allow accurate traceability of artifacts, and thus allows better update flows. For example, it will be easier to get different binaries for different configurations from different repositories, as long as they were created from the same recipe revision.

The package transfers, uploads, downloads, will also be more efficient, based on `revisions`. As long as a given revision exists on the server or in the cache, Conan will not transfer artifacts at all for that package.

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 different 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. Use a system installer, or create your own self-contained Conan executable, to not require Python in your system.
3. Running Conan from sources.

3.1 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.

Install Conan:

```
$ pip install conan
```

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.

3.1.1 Known installation issues with pip

When Conan is installed with **pip install --user <username>**, a new directory is usually 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 by restarting the session of the terminal or running the following command:

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

3.1.2 Update

If installed via `pip`, Conan version can be updated with:

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

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>/.conan2`).

3.2 Use a system installer or create a self-contained executable

There will be a number of existing installers in Conan downloads for OSX Brew, Debian, Windows, Linux Arch, that will not require Python first.

Note: These installers are not available at the moment of the 2.0 launch, but we will work to make them available after the launch. Please use the `pip install` or create your own self-contained executable using this instructions in the meantime.

If there is no installer for your platform, you can create your own Conan executable, with the `pyinstaller.py` utility in the repo. This process is able to create a self-contained Conan executable that contains all it needs, including the Python interpreter, so it wouldn't be necessary to have Python installed in the system.

You can do it with:

```
$ git clone https://github.com/conan-io/conan conan_src
$ cd conan_src
$ git checkout develop2 # or to the specific tag you want to
$ pip install -e .
$ python pyinstaller.py
```

It is important to install the dependencies and the project first with `pip install -e .` which configures the project as "editable", that is, to run from the current source folder. After creating the executable, it can be uninstalled with pip.

This has to run in the same platform that will be using the executable, pyinstaller does not cross-build. The resulting executable can be just copied and put in the system PATH of the running machine to be able to run Conan.

3.3 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 beta 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.

CHAPTER

FOUR

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.

4.1 Consuming packages

This section shows how to build your projects using Conan to manage your dependencies. We will begin with a 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 its 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*.

After that, we will introduce the concept of Conan build and host profiles and explain how you can use them to cross-compile your application to different platforms.

Then, in the "Introduction to versioning" we will learn about using different versions, defining requirements with version ranges, the concept of revisions and a brief introduction to lockfiles to achieve reproducibility of the dependency graph.

4.1.1 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.

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 examples2/tutorial/consuming_packages/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**

```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());

    return EXIT_SUCCESS;
}
```

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

Listing 2: **CMakeLists.txt**

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

find_package(ZLIB REQUIRED)

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

(continues on next page)

(continued from previous page)

```
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 allow users to define a configuration set for things like the 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 environment. 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 another profile is specified via the command line. An example of the output of this command for MacOS would be:

```
$ conan profile detect --force
Found apple-clang 14.0
apple-clang>=13, using the major as version
Detected profile:
[settings]
arch=x86_64
build_type=Release
compiler=apple-clang
compiler.cppstd=gnu17
compiler.libcxx=libc++
compiler.version=14
os=Macos
```

Note: A note about the detected C++ standard by Conan

Conan will always set the default C++ standard as the one that the detected compiler version uses by default, except for the case of macOS using apple-clang. In this case, for apple-clang>=11, it sets `compiler.cppstd=gnu17`. If you want to use a different C++ standard, you can edit the default profile file directly. First, get the location of the default profile using:

```
$ conan profile path default
/Users/user/.conan2/profiles/default
```

Then open and edit the file and set `compiler.cppstd` to the C++ standard you want to use.

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 *build*. To do that, run:

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

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

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

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, which are the files that define how libraries must be built, and the binaries that can be reused so we don't have to build from sources every time.

- Conan generated several files under the **build** 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 downloaded. 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 4: 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 5: Linux, macOS

```
$ cd build
$ cmake .. -DCMAKE_TOOLCHAIN_FILE=conan_toolchain.cmake -DCMAKE_BUILD_TYPE=Release
$ 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 Meson*
- Getting started with Autotools
- ...

4.1.2 Using build tools as Conan packages

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 of how to add a `tool_requires` to our project, and use 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 examples2/tutorial/consuming_packages/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 6: **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 7: **CMakeLists.txt**

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

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 the folder *build*. To do that, just run:

```
$ conan install . --output-folder=build --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/user/Documents/developer/conan/examples2/tutorial/consuming_
↪packages/tool_requires/conanfile.txt
```

(continues on next page)

(continued from previous page)

```
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
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 installed the new CMake version in the path.

Listing 8: Windows

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

Listing 9: Linux, macOS

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

Run `cmake` and check the version:

```
$ cmake --version
```

(continues on next page)

(continued from previous page)

```
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 currently active version now. Now you can build your project as you previously did, but this time Conan will use CMake 3.19.8 to build it:

Listing 10: 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 11: Linux, macOS

```
$ cmake .. -DCMAKE_TOOLCHAIN_FILE=conan_toolchain.cmake -DCMAKE_BUILD_TYPE=Release
$ 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 12: Windows

```
$ deactivate_conanbuild.bat
```

Listing 13: 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 [system_tools] in your profiles*.
- *Creating recipes for tool_requires: packaging build tools*.
- Using MinGW as tool_requires
- Using tool_requires in profiles
- Using conf to set a toolchain from a tool requires

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

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 examples2/tutorial/consuming_packages/different_configurations
```

So far, we built a simple CMake project that depended on the **zlib** library and learned about `tool_requires`, a special type or requirements for build-tools like CMake. In both cases, we did not specify anywhere that we wanted to build the application in *Release* or *Debug* mode, or if we wanted to link against *static* or *shared* libraries. That is because Conan, if not instructed otherwise, will use a default configuration declared in the 'default profile'. This default profile was created in the first example when we run the **conan profile detect** command. Conan stores this file in the **/profiles** folder, located in the Conan user home. You can check the contents of your default profile by running the **conan config home** command to get the location of the Conan user home and then showing the contents of the default profile in the **/profiles** folder:

```
$ conan config home
Current Conan home: /Users/tutorial_user/.conan2

# output the file contents
$ cat /Users/tutorial_user/.conan2/profiles/default
[settings]
os=Macos
arch=x86_64
compiler=apple-clang
compiler.version=14.0
compiler.libcxx=libc++
compiler.cppstd=gnu11
build_type=Release
[options]
[tool_requires]
[env]
```

As you can see, the profile has different sections. The [settings] section is the one that has information about things like the operating system, architecture, compiler, and build configuration. When you call a Conan command setting the --profile argument, Conan will take all the information from the profile and apply it to the packages you want to build or install. If you don't specify that argument it's equivalent to call it with --profile=default. These two commands will behave the same:

```
$ conan install . --build=missing
$ conan install . --build=missing --profile=default
```

You can store different profiles and use them to build for different settings. For example, to use a build_type=Debug, or adding a tool_requires to all the packages you build with that profile. One example of a *debug* profile could be:

Listing 14: <conan home>/profiles/debug

```
[settings]
os=Macos
arch=x86_64
compiler=apple-clang
compiler.version=14.0
compiler.libcxx=libc++
compiler.cppstd=gnu11
build_type=Debug
```

Modifying settings: use Debug configuration for the application and its dependencies

Using profiles is not the only way to set the configuration you want to use. You can also override the profile settings in the Conan command using the --settings argument. For example, you can build the project from the previous examples in *Debug* configuration instead of *Release*.

Before building, please check that we modified the source code from the previous example to show the build configuration the sources were built with:

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

int main(void) {
    ...
    #ifdef NDEBUG
    printf("Release configuration!\n");
    #else
    printf("Debug configuration!\n");
    #endif

    return EXIT_SUCCESS;
}
```

Now let's build our project for *Debug* configuration:

```
$ conan install . --output-folder=build --build=missing --settings=build_type=Debug
```

As we explained above, this is the equivalent of having *debug* profile and running these command using the --profile=debug argument instead of the --settings=build_type=Debug argument.

This **conan install** command will check if we already installed the required libraries (Zlib) in Debug configuration and install them otherwise. It will also set the build configuration in the conan_toolchain.cmake toolchain that the CMakeToolchain generator creates so that when we build the application it's built in *Debug* configuration. Now build your project as you did in the previous examples and check in the output how it was built in *Debug* configuration:

Listing 15: Windows

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

Listing 16: Linux, macOS

```
$ cd build
$ cmake .. -DCMAKE_TOOLCHAIN_FILE=conan_toolchain.cmake -DCMAKE_BUILD_TYPE=Debug
$ cmake --build .
$ ./compressor
Uncompressed size is: 233
Compressed size is: 147
ZLIB VERSION: 1.2.11
Debug configuration!
```

Modifying options: linking the application dependencies as shared libraries

So far, we have been linking *Zlib* statically in our application. That's because in the Zlib's Conan package there's an attribute set to build in that mode by default. We can change from **static** to **shared** linking by setting the `shared` option to `True` using the `--options` argument. To do so, please run:

Listing 17: Windows

```
$ conan install . --output-folder=build --build=missing --options=zlib/1.2.
↪11:shared=True
```

Doing this, Conan will install the *Zlib* shared libraries, generate the files to build with them and, also the necessary files to locate those dynamic libraries when running the application. Let's build the application again after configuring it to link *Zlib* as a shared library:

Listing 18: 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
```

Listing 19: Linux, Macos

```
$ cd build
```

(continues on next page)

(continued from previous page)

```
$ cmake .. -DCMAKE_TOOLCHAIN_FILE=conan_toolchain.cmake -DCMAKE_BUILD_TYPE=Release
$ cmake --build .
...
[100%] Built target compressor
```

Now, if you try to run the compiled executable you will see an error because the executable can't find the shared libraries for *Zlib* that we just installed.

Listing 20: Windows

```
$ Release\compressor.exe
(on a pop-up window) The code execution cannot proceed because zlib1.dll was not␣
↪found. Reinstalling the program may fix this problem.
```

Listing 21: Linux, Macos

```
$ ./compressor
./compressor: error while loading shared libraries: libz.so.1: cannot open shared␣
↪object file: No such file or directory
```

This is because 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*.

Conan provides a mechanism to define those variables and make it possible, for executables, to find and load these shared libraries. This mechanism is the `VirtualRunEnv` generator. If you check the output folder you will see that Conan generated a new file called `conanrun.sh/bat`. This is the result of automatically invoking that `VirtualRunEnv` generator when we activated the `shared` option when doing the **conan install**. This generated script will set the **PATH**, **LD_LIBRARY_PATH**, **DYLD_LIBRARY_PATH** and **DYLD_FRAMEWORK_PATH** environment variables so that executables can find the shared libraries.

Activate the virtual environment, and run the executables again:

Listing 22: Windows

```
$ conanrun.bat
$ Release\compressor.exe
Uncompressed size is: 233
Compressed size is: 147
...
```

Listing 23: Linux, macOS

```
$ source conanrun.sh
$ ./compressor
Uncompressed size is: 233
Compressed size is: 147
...
```

Just as in the previous example with the `VirtualBuildEnv` generator, when we run the `conanrun.sh/bat` script a deactivation script called `deactivate_conanrun.sh/bat` is created to restore the environment. Source or run it to do so:

Listing 24: Windows

```
$ deactivate_conanrun.bat
```

Listing 25: Linux, macOS

```
$ source deactivate_conanrun.sh
```

Difference between settings and options

You may have noticed that for changing between *Debug* and *Release* configuration we used a Conan **setting**, but when we set *shared* mode for our executable we used a Conan **option**. Please, note the difference between **settings** and **options**:

- **settings** are typically a project-wide configuration defined by the client machine. Things like the operating system, compiler or build configuration that will be common to several Conan packages and would not make sense to define one default value for only one of them. For example, it doesn't make sense for a Conan package to declare "Visual Studio" as a default compiler because that is something defined by the end consumer, and unlikely to make sense if they are working in Linux.
- **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 this is the linkage that should be used if consumers don't specify otherwise.

Read more

- *VirtualRunEnv reference*
- *Cross-compiling using –profile:build and –profile:host*
- Installing configurations with conan config install
- VS Multi-config
- Example about how settings and options influence the package id
- Using patterns for settings and options

4.1.4 Understanding the flexibility of using conanfile.py vs conanfile.txt

In the previous examples, we declared our dependencies (*Zlib* and *CMake*) in a *conanfile.txt* file. Let's have a look at that file:

Listing 26: **conanfile.txt**

```
[requires]
zlib/1.2.11

[tool_requires]
cmake/3.19.8

[generators]
CMakeDeps
CMakeToolchain
```

Using a *conanfile.txt* to build your projects using Conan it's enough for simple cases, but if you need more flexibility you should use a *conanfile.py* file where you can use Python code to make things such as adding requirements dynamically, changing options depending on other options or setting options for your requirements. Let's see an example on how to migrate to a *conanfile.py* and use some of those features.

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 examples2/tutorial/consuming_packages/conanfile_py
```

Check the contents of the folder and note that the contents are the same that in the previous examples but with a *conanfile.py* instead of a *conanfile.txt*.

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

Remember that in the previous examples the *conanfile.txt* had this information:

Listing 27: **conanfile.txt**

```
[requires]
zlib/1.2.11

[tool_requires]
cmake/3.19.8

[generators]
CMakeDeps
CMakeToolchain
```

We will translate that same information to a *conanfile.py*. This file is what is typically called a **"Conan recipe"**. It can be used for consuming packages, like in this case, and also to create packages. For our current case, it will define our requirements (both libraries and build tools) and logic to modify options and set how we want to consume those packages. In the case of using this file to create packages, it can define (among other things) how to download the package's source code, how to build the binaries from those sources, how to package the binaries, and information for future consumers on how to consume the package. We will explain how to use Conan recipes to create packages in the *Creating Packages* section later.

The equivalent of the *conanfile.txt* in form of Conan recipe could look like this:

Listing 28: **conanfile.py**

```
from conan import ConanFile


class CompressorRecipe(ConanFile):
    settings = "os", "compiler", "build_type", "arch"
    generators = "CMakeToolchain", "CMakeDeps"

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

    def build_requirements(self):
        self.tool_requires("cmake/3.19.8")
```

To create the Conan recipe we declared a new class that inherits from the `ConanFile` class. This class has different class attributes and methods:

- **settings** this class attribute defines the project-wide variables, like the compiler, its version, or the OS itself that may change when we build our project. This is related to how Conan manages binary compatibility as these values will affect the value of the **package ID** for Conan packages. We will explain how Conan uses this value to manage binary compatibility later.
- **generators** this class attribute specifies which Conan generators will be run when we call the **`conan install`** command. In this case, we added **CMakeToolchain** and **CMakeDeps** as in the *conanfile.txt*.
- **requirements()** in this method we can use the `self.requires()` and `self.tool_requires()` methods to declare all our dependencies (libraries and build tools).

You can check that running the same commands as in the previous examples will lead to the same results as before.

Listing 29: Windows

```
$ conan install . --output-folder=build --build=missing
$ cd build
$ conanbuild.bat
# 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
$ deactivate_conanbuild.bat
```

Listing 30: Linux, macOS

```
$ conan install . --output-folder build --build=missing
$ cd build
$ source conanbuild.sh
Capturing current environment in deactivate_conanbuildenv-release-x86_64.sh
Configuring environment variables
$ cmake .. -DCMAKE_TOOLCHAIN_FILE=conan_toolchain.cmake -DCMAKE_BUILD_TYPE=Release
$ 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
$ source deactivate_conanbuild.sh
```

So far we have achieved the same functionality we had using a *conanfile.txt*, let's see how we can take advantage of the capabilities of the *conanfile.py* to define the project structure we want to follow and also to add some logic using Conan settings and options.

Conditional requirements using a conanfile.py

You could add some logic to the *requirements() method* to add or remove requirements conditionally. Imagine, for example, that you want to add an additional dependency in Windows or that you want to use the system's CMake installation instead of using the Conan *tool_requires*:

Listing 31: **conanfile.py**

```
from conan import ConanFile


class CompressorRecipe(ConanFile):
    # Binary configuration
    settings = "os", "compiler", "build_type", "arch"
    generators = "CMakeToolchain", "CMakeDeps"

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

        # Add base64 dependency for Windows
        if self.settings.os == "Windows":
            self.requires("base64/0.4.0")

    def build_requirements(self):
        # Use the system's CMake for Windows
        if self.settings.os != "Windows":
            self.tool_requires("cmake/3.19.8")
```

Use the layout() method

In the previous examples, every time we executed a *conan install* command we had to use the *–output-folder argument* to define where we wanted to create the files that Conan generates. Also, note that we used a different folder when building in Windows or in Linux/macOS depending if we were using a multi-config CMake generator or not. You can define this directly in the *conanfile.py* inside the *layout()* method and make it work for every platform without adding more changes:

Listing 32: **conanfile.py**

```
from conan import ConanFile


class CompressorRecipe(ConanFile):
    settings = "os", "compiler", "build_type", "arch"
    generators = "CMakeToolchain", "CMakeDeps"

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

    def build_requirements(self):
        self.tool_requires("cmake/3.19.8")

    def layout(self):
        # We make the assumption that if the compiler is msvc the
        # CMake generator is multi-config
        if self.settings.get_safe("compiler") == "msvc":
            multi = True
```

(continues on next page)

(continued from previous page)

```
        else:
            multi = False

        self.folders.build = "build" if multi else f"build/{str(self.settings.build_
↪type)}"
        self.folders.generators = "build"
```

As you can see, we defined two different attributes for the Conanfile in the *layout()* method:

- **self.folders.build** is the folder where the resulting binaries will be placed. The location depends on the type of CMake generator. For multi-config, they will be located in a dedicated folder inside the build folder, while for single-config, they will be located directly in the build folder.
- **self.folders.generators** is the folder where all the auxiliary files generated by Conan (CMake toolchain and cmake dependencies files) will be placed.

Note that the definitions of the folders is different if it is a multi-config generator (like Visual Studio), or a single-config generator (like Unix Makefiles). In the first case, the folder is the same irrespective of the build type, and the build system will manage the different build types inside that folder. But single-config generators like Unix Makefiles, must use a different folder for each different configuration (as a different build_type Release/Debug). In this case we added a simple logic to consider multi-config if the compiler name is *msvc*.

Check that running the same commands as in the previous examples without the *–output-folder* argument will lead to the same results as before:

Listing 33: Windows

```
$ conan install . --build=missing
$ cd build
$ conanbuild.bat
# 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
$ deactivate_conanbuild.bat
```

Listing 34: Linux, macOS

```
$ conan install . --build=missing
$ cd build
$ source conanbuild.sh
Capturing current environment in deactivate_conanbuildenv-release-x86_64.sh
Configuring environment variables
$ cmake .. -DCMAKE_TOOLCHAIN_FILE=conan_toolchain.cmake -DCMAKE_BUILD_TYPE=Release
$ cmake --build .
...
Building with CMake version: 3.19.8
...
```

(continues on next page)

(continued from previous page)

```
[100%] Built target compressor

$ ./Release/compressor
Uncompressed size is: 233
Compressed size is: 147
ZLIB VERSION: 1.2.11
$ source deactivate_conanbuild.sh
```

There's no need to always write this logic in the *conanfile.py*. There are some pre-defined layouts you can import and directly use in your recipe. For example, for the CMake case, there's a *cmake_layout()* already defined in Conan:

Listing 35: **conanfile.py**

```
from conan import ConanFile
from conan.tools.cmake import cmake_layout


class CompressorRecipe(ConanFile):
    settings = "os", "compiler", "build_type", "arch"
    generators = "CMakeToolchain", "CMakeDeps"

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

    def build_requirements(self):
        self.tool_requires("cmake/3.19.8")

    def layout(self):
        cmake_layout(self)
```

Use the validate() method to raise an error for non-supported configurations

The *validate() method* is evaluated when Conan loads the *conanfile.py* and you can use it to perform checks of the input settings. If, for example, your project does not support *armv8* architecture on macOS you can raise the *ConanInvalidConfiguration* exception to make Conan return with a special error code. This will indicate that the configuration used for settings or options is not supported.

Listing 36: **conanfile.py**

```
...
from conan.errors import ConanInvalidConfiguration

class CompressorRecipe(ConanFile):
    ...

    def validate(self):
        if self.settings.os == "Macos" and self.settings.arch == "armv8":
            raise ConanInvalidConfiguration("ARM v8 not supported")
```

Read more

- *Using "cmake_layout" + "CMakeToolchain" + "CMakePresets feature" to build your project*.
- *Understanding the Conan Package layout*.

- Importing resource files in the generate() method
- Conditional generators in configure()

4.1.5 How to cross-compile your applications using Conan: host and build contexts

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

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

In the previous examples, we learned how to use a *conanfile.py* or *conanfile.txt* to build an application that compresses strings using the *Zlib* and *CMake* Conan packages. Also, we explained that you can set information like the operating system, compiler or build configuration in a file called the Conan profile. You can use that profile as an argument (**--profile**) to invoke the **conan install**. We also explained that not specifying that profile is equivalent to using the **--profile=default** argument.

For all those examples, we used the same platform for building and running the application. But, what if you want to build the application on your machine running Ubuntu Linux and then run it on another platform like a Raspberry Pi? Conan can model that case using two different profiles, one for the machine that **builds** the application (Ubuntu Linux) and another for the machine that **runs** the application (Raspberry Pi). We will explain this "two profiles" approach in the next section.

Conan two profiles model: build and host profiles

Even if you specify only one **--profile** argument when invoking Conan, Conan will internally use two profiles. One for the machine that **builds** the binaries (called the **build** profile) and another for the machine that **runs** those binaries (called the **host** profile). Calling this command:

```
$ conan install . --build=missing --profile=someprofile
```

Is equivalent to:

```
$ conan install . --build=missing --profile:host=someprofile --profile:build=default
```

As you can see we used two new arguments:

- `profile:host`: This is the profile that defines the platform where the built binaries will run. For our string compressor application this profile would be the one applied for the *Zlib* library that will run in a **Raspberry Pi**.
- `profile:build`: This is the profile that defines the platform where the binaries will be built. For our string compressor application, this profile would be the one used by the *CMake* tool that will compile it on the **Ubuntu Linux** machine.

Note that when you just use one argument for the profile `--profile` is equivalent to `--profile:host`. If you don't specify the `--profile:build` argument, Conan will use the *default* profile internally.

So, if we want to build the compressor application in the Ubuntu Linux machine but run it in a Raspberry Pi, we should use two different profiles. For the **build** machine we could use the default profile, that in our case looks like this:

Listing 37: <conan home>/profiles/default

```
[settings]
os=Linux
arch=x86_64
```

(continues on next page)

(continued from previous page)

```
build_type=Release
compiler=gcc
compiler.cppstd=gnu14
compiler.libcxx=libstdc++11
compiler.version=9
```

And the profile for the Raspberry Pi that is the **host** machine:

Listing 38: <local folder>/profiles/raspberry

```
[settings]
os=Linux
arch=armv7hf
compiler=gcc
build_type=Release
compiler.cppstd=gnu14
compiler.libcxx=libstdc++11
compiler.version=9
[buildenv]
CC=arm-linux-gnueabihf-gcc-9
CXX=arm-linux-gnueabihf-g++-9
LD=arm-linux-gnueabihf-ld
```

Important: Please, take into account that in order to build this example successfully, you should have installed a toolchain that includes the compiler and all the tools to build the application for the proper architecture. In this case the host machine is a Raspberry Pi 3 with *armv7hf* architecture operating system and we have the *arm-linux-gnueabihf* toolchain installed in the Ubuntu machine.

If you have a look at the *raspberry* profile, there is a section named `[buildenv]`. This section is used to set the environment variables that are needed to build the application. In this case we declare the `CC`, `CXX` and `LD` variables pointing to the cross-build toolchain compilers and linker, respectively. Adding this section to the profile will invoke the VirtualBuildEnv generator everytime we do a **conan install**. This generator will add that environment information to the `conanbuild.sh` script that we will source before building with CMake so that it can use the cross-build toolchain.

Build and host contexts

Now that we have our two profiles prepared, let's have a look at our *conanfile.py*:

Listing 39: **conanfile.py**

```python
from conan import ConanFile
from conan.tools.cmake import cmake_layout

class CompressorRecipe(ConanFile):
    settings = "os", "compiler", "build_type", "arch"
    generators = "CMakeToolchain", "CMakeDeps"

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

    def build_requirements(self):
        self.tool_requires("cmake/3.19.8")
```

(continues on next page)

(continued from previous page)

```
    def layout(self):
        cmake_layout(self)
```

As you can see, this is practically the same *conanfile.py* we used in the *previous example*. We will require **zlib/1.2.11** as a regular dependency and **cmake/3.19.8** as a tool needed for building the application.

We will need the application to build for the Raspberry Pi with the cross-build toolchain and also link the **zlib/1.2.11** library built for the same platform. On the other side, we need the **cmake/3.19.8** binary to run in Ubuntu Linux. Conan manages this internally in the dependency graph differentiating between what we call the "build context" and the "host context":

- The **host context** is populated with the root package (the one specified in the **conan install** or **conan create** command) and all its requirements added via `self.requires()`. In this case, this includes the compressor application and the **zlib/1.2.11** dependency.
- The **build context** contains the tool requirements used in the build machine. This category typically includes all the developer tools like CMake, compilers and linkers. In this case, this includes the **cmake/3.19.8** tool.

These contexts define how Conan will manage each one of the dependencies. For example, as **zlib/1.2.11** belongs to the **host context**, the `[buildenv]` build environment we defined in the **raspberry** profile (profile host) will only apply to the **zlib/1.2.11** library when building and won't affect anything that belongs to the **build context** like the **cmake/3.19.8** dependency.

Now, let's build the application. First, call **conan install** with the profiles for the build and host platforms. This will install the **zlib/1.2.11** dependency built for *armv7hf* architecture and a **cmake/3.19.8** version that runs for 64-bit architecture.

```
$ conan install . --build missing -pr:b=default -pr:h=./profiles/raspberry
```

Then, let's call CMake to build the application. As we did in the previous example we have to activate the **build environment** running `source Release/generators/conanbuild.sh`. That will set the environment variables needed to locate the cross-build toolchain and build the application.

```
$ cd build
$ source Release/generators/conanbuild.sh
Capturing current environment in deactivate_conanbuildenv-release-armv7hf.sh
Configuring environment variables
$ cmake .. -DCMAKE_TOOLCHAIN_FILE=Release/generators/conan_toolchain.cmake -DCMAKE_
↪BUILD_TYPE=Release
$ cmake --build .
...
-- Conan toolchain: C++ Standard 14 with extensions ON
-- The C compiler identification is GNU 9.4.0
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /usr/bin/arm-linux-gnueabihf-gcc-9 - skipped
-- Detecting C compile features
-- Detecting C compile features - done    [100%] Built target compressor
...
$ source Release/generators/deactivate_conanbuild.sh
```

You could check that we built the application for the correct architecture by running the `file` Linux utility:

```
$ file compressor
compressor: ELF 32-bit LSB shared object, ARM, EABI5 version 1 (SYSV), dynamically
linked, interpreter /lib/ld-linux-armhf.so.3,
```

(continues on next page)

(continued from previous page)

```
BuildID[sha1]=2a216076864a1b1f30211debf297ac37a9195196, for GNU/Linux 3.2.0, not
stripped
```

Read more

- *Cross building to Android with the NDK*
- *VirtualBuildEnv reference*
- Cross-build using a tool_requires
- How to require test frameworks like gtest: using `test_requires`
- Using Conan to build for iOS

4.1.6 Introduction to versioning

So far we have been using requires with fixed versions like `requires = "zlib/1.2.12"`. But sometimes dependencies evolve, new versions are released and consumers want to update to those versions as easy as possible.

It is always possible to edit the `conanfiles` and explicitly update the versions to the new ones, but there are mechanisms in Conan to allow such updates without even modifying the recipes.

Version ranges

A `requires` can express a dependency to a certain range of versions for a given package, with the syntax `pkgname/[version-range-expression]`. Let's see an example, 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 examples2/tutorial/consuming_packages/versioning
```

We can see that we have there:

Listing 40: **conanfile.py**

```python
from conan import ConanFile


class CompressorRecipe(ConanFile):
    settings = "os", "compiler", "build_type", "arch"
    generators = "CMakeToolchain", "CMakeDeps"

    def requirements(self):
        self.requires("zlib/[~1.2]")
```

That `requires` contains the expression `zlib/[~1.2]`, which means "approximately" `1.2` version, that means, it can resolve to any `zlib/1.2.8`, `zlib/1.2.11` or `zlib/1.2.12`, but it will not resolve to something like `zlib/1.3.0`. Among the available matching versions, a version range will always pick the latest one.

If we do a **conan install**, we would see something like:

```
$ conan install .

Graph root
    conanfile.py: .../conanfile.py
Requirements
    zlib/1.2.12#87a7211557b6690ef5bf7fc599dd8349 - Downloaded
Resolved version ranges
    zlib/[~1.2]: zlib/1.2.12
```

If we tried instead to use `zlib/[<1.2.12]`, that means that we would like to use a version lower than `1.2.12`, but that one is excluded, so the latest one to satisfy the range would be `zlib/1.2.11`:

```
$ conan install .

Resolved version ranges
    zlib/[<1.2.12]: zlib/1.2.11
```

The same applies to other type of requirements, like `tool_requires`. If we add now to the recipe:

Listing 41: **conanfile.py**

```python
from conan import ConanFile


class CompressorRecipe(ConanFile):
    settings = "os", "compiler", "build_type", "arch"
    generators = "CMakeToolchain", "CMakeDeps"

    def requirements(self):
        self.requires("zlib/[~1.2]")

    def build_requirements(self):
        self.tool_requires("cmake/[>3.10]")
```

Then we would see it resolved to the latest available CMake package, with at least version `3.11`:

```
$ conan install .
...
Graph root
    conanfile.py: .../conanfile.py
Requirements
    zlib/1.2.12#87a7211557b6690ef5bf7fc599dd8349 - Cache
Build requirements
    cmake/3.19.8#f305019023c2db74d1001c5afa5cf362 - Downloaded
Resolved version ranges
    cmake/[>3.10]: cmake/3.19.8
    zlib/[~1.2]: zlib/1.2.12
```

Revisions

What happens when a package creator does some change to the package recipe or to the source code, but they don't bump the `version` to reflect those changes? Conan has an internal mechanism to keep track of those modifications, and it is called the **revisions**.

The recipe revision is the hash that can be seen together with the package name and version in the form `pkgname/version#recipe_revision` or `pkgname/version@user/channel#recipe_revision`. The recipe revision is a hash of the contents of the recipe and the source code. So if something changes either in the recipe, its associated files or in the source code that this recipe is packaging, it will create a new recipe revision.

You can list existing revisions with the **conan list** command:

```
conan list zlib/1.2.12#* -r=conanv2

conanv2:
zlib
    zlib/1.2.12#0de8ff7f99079cd07341311c9ead89a2 (2022-12-12 11:39:43 UTC)
    zlib/1.2.12#6758146baf425dc62ecc5246a1e955e4 (2022-08-25 07:45:24 UTC)
    zlib/1.2.12#87a7211557b6690ef5bf7fc599dd8349 (2022-04-21 11:01:59 UTC)
```

Revisions always resolve to the latest (chronological order of creation or upload to the server) revision. Though it is not a common practice, it is possible to explicitly pin a given recipe revision directly in the `conanfile`, like:

```
def requirements(self):
    self.requires("zlib/1.2.12#87a7211557b6690ef5bf7fc599dd8349")
```

This mechanism can however be tedious to maintain and update when new revisions are created, so probably in the general case, this shouldn't be done.

Lockfiles

The usage of version ranges, and the possibility of creating new revisions of a given package without bumping the version allows to do automatic faster and more convenient updates, without need to edit recipes.

But in some occassions, there is also a need to provide an immutable and reproducible set of dependencies. This process is known as "locking", and the mechanism to allow it is "lockfile" files. A lockfile is a file that contains a fixed list of dependencies, specifying the exact version and exact revision. So, for example, a lockfile will never contain a version range with an expression, but only pinned dependencies.

A lockfile can be seen as a snapshot of a given dependency graph at some point in time. Such snapshot must be "realizable", that is, it needs to be a state that can be actually reproduced from the conanfile recipes. And this lockfile can be used at a later point in time to force that same state, even if there are new created package versions.

Let's see lockfiles in action. First, let's pin the dependency to `zlib/1.2.11` in our example:

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

And let's capture a lockfile:

```
conan lock create .

-------- Computing dependency graph ----------
Graph root
    conanfile.py: .../conanfile.py
Requirements
    zlib/1.2.11#4524fcdd41f33e8df88ece6e755a5dcc - Cache

Generated lockfile: .../conan.lock
```

Let's see what the lockfile `conan.lock` contains:

```
{
    "version": "0.5",
```

(continues on next page)

(continued from previous page)

```
    "requires": [
        "zlib/1.2.11#4524fcdd41f33e8df88ece6e755a5dcc%1650538915.154"
    ],
    "build_requires": [],
    "python_requires": []
}
```

Now, let's restore the original `requires` version range:

```
def requirements(self):
    self.requires("zlib/[~1.2]")
```

And run **conan install .**, which by default will find the `conan.lock`, and run the equivalent **conan install . --lockfile=conan.lock**

```
conan install .

Graph root
    conanfile.py: .../conanfile.py
Requirements
    zlib/1.2.11#4524fcdd41f33e8df88ece6e755a5dcc - Cache
```

Note how the version range is no longer resolved, and it doesn't get the `zlib/1.2.12` dependency, even if it is the allowed range `zlib/[~1.2]`, because the `conan.lock` lockfile is forcing it to stay in `zlib/1.2.11` and that exact revision too.

Read more

- *Introduction to Versioning*

4.2 Creating packages

This section shows how to create Conan packages using a Conan recipe. We begin by creating a basic Conan recipe to package a simple C++ library that you can scaffold using the **conan new** command. Then, we will explain the different methods that you can define inside a Conan recipe and the things you can do inside them:

- Using the `source()` method to retrieve sources from external repositories and apply patches to those sources.
- Add requirements to your Conan packages inside the `requirements()` method.
- Use the `generate()` method to prepare the package build, and customize the toolchain.
- Configure settings and options in the `configure()` and `config_options()` methods and how they affect the packages' binary compatibility.
- Use the `build()` method to customize the build process and launch the tests for the library you are packaging.
- Select which files will be included in the Conan package using the `package()` method.
- Define the package information in the `package_info()` method so that consumers of this package can use it.
- Use a *test_package* to test that the Conan package can be consumed correctly.

After this walkthrough around some Conan recipe methods, we will explain some peculiarities of different types of Conan packages like, for example, header-only libraries, packages for pre-built binaries, packaging tools for building other packages or packaging your own applications.

4.2.1 Create your first Conan package

In previous sections, we *consumed* Conan packages (like the *Zlib* one), first using a *conanfile.txt* and then with a *conanfile.py*. But a *conanfile.py* recipe file is not only meant to consume other packages, it can be used to create your own packages as well. In this section, we explain how to create a simple Conan package with a *conanfile.py* recipe and how to use Conan commands to build those packages from sources.

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 (such as VS, Meson, Autotools, and even your own) to do that, without any dependency on CMake.

Use the **conan new** command to create a "Hello World" C++ library example project:

```
$ conan new cmake_lib -d name=hello -d version=1.0
```

This will create a Conan package project with the following structure.

```
.
├── CMakeLists.txt
├── conanfile.py
├── include
│   └── hello.h
├── src
│   └── hello.cpp
└── test_package
    ├── CMakeLists.txt
    ├── conanfile.py
    └── 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.
- **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 conan import ConanFile
from conan.tools.cmake import CMakeToolchain, CMake, cmake_layout


class helloRecipe(ConanFile):
    name = "hello"
    version = "1.0"

    # 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 package here>"
```

(continues on next page)

(continued from previous page)

```
    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/*", "include/*"

    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 the different sections of the recipe briefly:

First, you can see the **name and version** of the Conan package defined:

- `name`: a string, with a minimum of 2 and a maximum of 100 **lowercase** characters that defines the package name. It should start with alphanumeric or underscore and can contain alphanumeric, underscore, +, ., - characters.
- `version`: 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.

Then you can see, some attributes defining **metadata**. These are optional but recommended and define things like a short `description` for the package, the `author` of the packaged library, the `license`, the `url` for the package repository, and the `topics` that the package is related to.

After that, there is a section related with the binary configuration. This section defines the valid settings and options for the package. As we explained in the *consuming packages section*:

- `settings` are project-wide configuration that cannot be defaulted in recipes. Things like the operating system, compiler or build configuration that will be common to several Conan packages
- `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.

After that, the `exports_sources` attribute is set to define which sources are part of the Conan package. These are the sources for the library you want to package. In this case the sources for our "hello" library.

Then, several methods are declared:

- 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 `layout()` method declares the locations where we expect to find the source files and also those where we want to save the generated files during the build process. Things like the folder for the generated binaries or all the files that the Conan generators create in the `generate()` method. In this case, as our project uses CMake as the build system, we call to `cmake_layout()`. Calling this function will set the expected locations for a CMake project.
- 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 an equivalent using the *copy() tool* in the `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. This is generic information about the current package, and is available to the consumers irrespective of the build system they are using and irrespective of the build system we have used in the `build()` method

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 `conanfile.py`, and its source code including build scripts, that depends on the package being created, and builds and executes a small application that requires the library in the package.
- It doesn't belong in the package. It only exists 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 .
-------- Exporting the recipe ----------
hello/1.0: Exporting package recipe
...
[ 50%] Building CXX object CMakeFiles/example.dir/src/example.cpp.o
[100%] Linking CXX executable example
[100%] Built target example

-------- Testing the package: Running test() ----------
hello/1.0 (test package): Running test()
hello/1.0 (test package): RUN: ./example
hello/1.0: Hello World Release!
  hello/1.0: __x86_64__ defined
```

(continues on next page)

(continued from previous page)

```
  hello/1.0: __cplusplus199711
  hello/1.0: __GNUC__4
  hello/1.0: __GNUC_MINOR__2
  hello/1.0: __clang_major__13
  hello/1.0: __clang_minor__1
  hello/1.0: __apple_build_version__13160021
...
```

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/1.0` package starts, calling the `generate()`, `build()` and `package()` methods. This creates the binary package in the Conan cache.
- Conan then 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 hello
Local Cache:
  hello
    hello/1.0
```

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 Debug configuration or to build the hello library as shared:

```
$ conan create . -s build_type=Debug
...
hello/1.0: Hello World Debug!

$ conan create . -o hello/1.0:shared=True
...
hello/1.0: 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:

```
# list the binary built for the hello/1.0 package
# latest is a placeholder to show the package that is the latest created
$ conan list hello/1.0#:*
Local Cache:
hello
  hello/1.0#fa5f6b17d0adc4de6030c9ab71cdbede (2022-12-22 17:32:19 UTC)
    PID: 6679492451b5d0750f14f9024fdbf84e19d2941b (2022-12-22 17:32:20 UTC)
      settings:
        arch=x86_64
        build_type=Release
        compiler=apple-clang
        compiler.cppstd=gnu11
        compiler.libcxx=libc++
        compiler.version=14
        os=Macos
      options:
```

(continues on next page)

(continued from previous page)

```
      fPIC=True
      shared=True
  PID: b1d267f77ddd5d10d06d2ecf5a6bc433fbb7eeed (2022-12-22 17:31:59 UTC)
    settings:
      arch=x86_64
      build_type=Release
      compiler=apple-clang
      compiler.cppstd=gnu11
      compiler.libcxx=libc++
      compiler.version=14
      os=Macos
    options:
      fPIC=True
      shared=False
  PID: d15c4f81b5de757b13ca26b636246edff7bdbf24 (2022-12-22 17:32:14 UTC)
    settings:
      arch=x86_64
      build_type=Debug
      compiler=apple-clang
      compiler.cppstd=gnu11
      compiler.libcxx=libc++
      compiler.version=14
      os=Macos
    options:
      fPIC=True
```

Now that we have created a simple Conan package, we will explain each of the methods of the Conanfile in more detail. You will learn how to modify those methods to achieve things like retrieving the sources from an external repository, adding dependencies to our package, customising our toolchain and much more.

A note about the Conan cache

When you did the **`conan create`** command, the build of your package did not take place in your local folder but in other folder inside the *Conan cache*. This cache is located in the user home folder under the `.conan2` folder. Conan will use the `~/.conan2` folder to store the built packages and also different configuration files. You already used the **`conan list`** command to list the recipes and binaries stored in the local cache.

Read more

- *Conan list command reference*.
- Create your first Conan package with Autotools.
- Create your first Conan package with Meson.
- Create your first Conan package with Visual Studio.

4.2.2 Handle sources in packages

In the *previous tutorial section* we created a Conan package for a "Hello World" C++ library. We used the `exports_sources` attribute of the Conanfile to declare the location of the sources for the library. This method is the simplest way to define the location of the source files when they are in the same folder as the Conanfile. However, sometimes the source files are stored in a repository or a file in a remote server, and not in the same location as the Conanfile. In this section, we will modify the recipe we created previously by adding a `source()` method and explain how to:

- Retrieve the sources from a *zip* file stored in a remote repository.
- Retrieve the sources from a branch of a *git* repository.

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

```
$ git clone https://github.com/conan-io/examples2.git
$ cd examples2/tutorial/creating_packages/handle_sources
```

The structure of the project is the same as the one in the previous example but without the library sources:

```
.
├── CMakeLists.txt
├── conanfile.py
└── test_package
    ├── CMakeLists.txt
    ├── conanfile.py
    └── src
        └── example.cpp
```

Sources from a *zip* file stored in a remote repository

Let's have a look at the changes in the *conanfile.py*:

```
from conan import ConanFile
from conan.tools.cmake import CMakeToolchain, CMake, cmake_layout
from conan.tools.files import get


class helloRecipe(ConanFile):
    name = "hello"
    version = "1.0"

    ...

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

    def source(self):
        get(self, "https://github.com/conan-io/libhello/archive/refs/heads/main.zip",
                  strip_root=True)

    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()
```

(continues on next page)

(continued from previous page)

```
    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"]
```

As you can see, the recipe is the same but instead declaring the `exports_sources` attribute as we did previously:

```
exports_sources = "CMakeLists.txt", "src/*", "include/*"
```

We declare a `source()` method with this information:

```
def source(self):
    get(self, "https://github.com/conan-io/libhello/archive/refs/heads/main.zip",
              strip_root=True)
```

We used the *conan.tools.files.get()* tool that will first **download** the *zip* file from the URL that we pass as an argument and then **unzip** it. Note that we pass the `strip_root=True` argument so that if all the unzipped contents are in a single folder, all the contents are moved to the parent folder (check the *conan.tools.files.unzip()* reference for more details).

The contents of the zip file are the same as the sources we previously had beside the Conan recipe, so if you do a **conan create** the results will be the same as before.

```
$ conan create .

...

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

Installing (downloading, building) binaries...
hello/1.0: Calling source() in /Users/user/.conan2/p/0fcb5ffd11025446/s/.
Downloading update_source.zip

hello/1.0: Unzipping 3.7KB
Unzipping 100 %
hello/1.0: Copying sources to build folder
hello/1.0: Building your package in /Users/user/.conan2/p/tmp/369786d0fb355069/b

...

-------- Testing the package: Running test() ----------
hello/1.0 (test package): Running test()
hello/1.0 (test package): RUN: ./example
hello/1.0: Hello World Release!
hello/1.0: __x86_64__ defined
hello/1.0: __cplusplus199711
hello/1.0: __GNUC__4
hello/1.0: __GNUC_MINOR__2
hello/1.0: __clang_major__13
```

(continues on next page)

(continued from previous page)

```
hello/1.0: __clang_minor__1
hello/1.0: __apple_build_version__13160021
```

Please, check the highlighted lines with the messages about the download and unzip operation.

Sources from a branch in a *git* repository

Now, let's modify the `source()` method to bring the sources from a *git* repository instead of a *zip* file. We show just the relevant parts:

```
...

from conan.tools.scm import Git


class helloRecipe(ConanFile):
    name = "hello"
    version = "1.0"

    ...

    def source(self):
        git = Git(self)
        git.clone(url="https://github.com/conan-io/libhello.git", target=".")

    ...
```

Here, we use the *conan.tools.scm.Git()* tool. The `Git` class implements several methods to work with *git* repositories. In this case, we call the clone method to clone the https://github.com/conan-io/libhello.git repository in the default branch using the same folder for cloning the sources instead of a subfolder (passing the `target="."` argument).

If we wanted to checkout a commit or tag in the repository we could use the `checkout()` method of the Git tool:

```
def source(self):
    git = Git(self)
    git.clone(url="https://github.com/conan-io/libhello.git", target=".")
    git.checkout("<branch name>, <tag> or <commit hash>")
```

For more information about the `Git` class methods, please check the *conan.tools.scm.Git()* reference.

Note that it's also possible to run other commands by invoking the `self.run()` method.

Using the conandata.yml file

We can write a file named `conandata.yml` in the same folder of the `conanfile.py`. This file will be automatically exported and parsed by Conan and we can read that information from the recipe. This is handy for example to extract the URLs of the external sources repositories, zip files etc. This is an example of `conandata.yml`:

```
sources:
  "1.0":
    url: "https://github.com/conan-io/libhello/archive/refs/heads/main.zip"
    sha256: "7bc71c682895758a996ccf33b70b91611f51252832b01ef3b4675371510ee466"
    strip_root: true
  "1.1":
```

(continues on next page)

(continued from previous page)

```
    url: ...
    sha256: ...
```

The recipe doesn't need to be modified for each version of the code. We can pass all the `keys` of the specified version (`url`, `sha256`, and `strip_root`) as arguments to the `get` function, that, in this case, allow us to verify that the downloaded zip file has the correct `sha256`. So we could modify the source method to this:

```
def source(self):
    get(self, **self.conan_data["sources"][self.version])
    # Similar to:
    # data = self.conan_data["sources"][self.version]
    # get(self, data["url"], sha256=data["sha256"], strip_root=data["strip_root"])
```

Read more

- *Patching sources*
- Advanced git repository handling (implement the "scm feature")
- ...

See also:

- *source() method reference*

4.2.3 Add dependencies to packages

In the *previous tutorial section* we created a Conan package for a "Hello World" C++ library. We used the *conan.tools.scm.Git()* tool to retrieve the sources from a git repository. So far, the package does not have any dependency on other Conan packages. Let's explain how to add a dependency to our package in a very similar way that we did in the *consuming packages section*. We will add some fancy colour output to our "Hello World" library using the fmt library.

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

```
$ git clone https://github.com/conan-io/examples2.git
$ cd examples2/tutorial/creating_packages/add_requires
```

You will notice some changes in the *conanfile.py* file from the previous recipe. Let's check the relevant parts:

```
...
from conan.tools.build import check_max_cppstd, check_min_cppstd
...

class helloRecipe(ConanFile):
    name = "hello"
    version = "1.0"

    ...
    generators = "CMakeDeps"
    ...

    def validate(self):
        check_min_cppstd(self, "11")
        check_max_cppstd(self, "14")
```

(continues on next page)

(continued from previous page)

```
    def requirements(self):
        self.requires("fmt/8.1.1")

    def source(self):
        git = Git(self)
        git.clone(url="https://github.com/conan-io/libhello.git", target=".")
        # Please, be aware that using the head of the branch instead of an immutable␣
↪tag
        # or commit is not a good practice in general
        git.checkout("require_fmt")
```

- First, we set the generators class attribute to make Conan invoke the *CMakeDeps* generator. This was not needed in the previous recipe as we did not have dependencies. CMakeDeps will generate all the config files CMake needs to find the fmt library.
- Next, we use the *requires()* method to add the fmt dependency to our package.
- Also, check that we added an extra line in the *source()* method. We use the *Git().checkout* method to checkout the source code in the require_fmt branch. This branch contains the changes in the source code to add colours to the library messages, and also in the CMakeLists.txt to declare that we are using the fmt library.
- Finally, note we added the *validate()* method to the recipe. We already used this method in the *consuming packages section* to raise an error for non-supported configurations. Here, we call the *check_min_cppstd()* and *check_max_cppstd()* to check that we are using at least C++11 and at most C++14 standards in our settings.

You can check the new sources, using the fmt library in the require_fmt. You will see that the hello.cpp file adds colours to the output messages:

```
#include <fmt/color.h>

#include "hello.h"

void hello(){
    #ifdef NDEBUG
    fmt::print(fg(fmt::color::crimson) | fmt::emphasis::bold, "hello/1.0: Hello World␣
↪Release!\n");
    #else
    fmt::print(fg(fmt::color::crimson) | fmt::emphasis::bold, "hello/1.0: Hello World␣
↪Debug!\n");
    #endif
    ...
```

Let's build the package from sources with the current default configuration, and then let the test_package folder test the package. You should see the output messages with colour now:

```
$ conan create . --build=missing
-------- Exporting the recipe ----------
...
-------- Testing the package: Running test() ----------
hello/1.0 (test package): Running test()
hello/1.0 (test package): RUN: ./example
hello/1.0: Hello World Release!
  hello/1.0: __x86_64__ defined
  hello/1.0: __cplusplus 201103
  hello/1.0: __GNUC__ 4
  hello/1.0: __GNUC_MINOR__ 2
```

(continues on next page)

(continued from previous page)

```
hello/1.0: __clang_major__ 13
hello/1.0: __clang_minor__ 1
hello/1.0: __apple_build_version__ 13160021
```

Read more

- *Reference for requirements() method*.
- *Introduction to versioning*.

4.2.4 Preparing the build

In the *previous tutorial section*, we added the fmt requirement to our Conan package to provide colour output to our "Hello World" C++ library. In this section, we focus on the `generate()` method of the recipe. The aim of this method generating all the information that could be needed while running the build step. That means things like:

- Write files to be used in the build step, like *scripts* that inject environment variables, files to pass to the build system, etc.
- Configuring the toolchain to provide extra information based on the settings and options or removing information from the toolchain that Conan generates by default and may not apply for certain cases.

We explain to use this method for a simple example based on the previous tutorial section. We add a *with_fmt* option to the recipe, depending on the value we require the *fmt* library or not. We use the *generate()* method to modify the toolchain so that it passes a variable to CMake so that we can conditionally add that library and use *fmt* or not in the source code.

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

```
$ git clone https://github.com/conan-io/examples2.git
$ cd examples2/tutorial/creating_packages/preparing_the_build
```

You will notice some changes in the *conanfile.py* file from the previous recipe. Let's check the relevant parts:

```
...
from conan.tools.build import check_max_cppstd, check_min_cppstd
...

class helloRecipe(ConanFile):
    name = "hello"
    version = "1.0"

    ...
    options = {"shared": [True, False],
               "fPIC": [True, False],
               "with_fmt": [True, False]}

    default_options = {"shared": False,
                       "fPIC": True,
                       "with_fmt": True}
    ...

    def validate(self):
        if self.options.with_fmt:
            check_min_cppstd(self, "11")
```

(continues on next page)

(continued from previous page)

```
            check_max_cppstd(self, "14")

    def source(self):
        git = Git(self)
        git.clone(url="https://github.com/conan-io/libhello.git", target=".")
        # Please, be aware that using the head of the branch instead of an immutable␣
↪tag
        # or commit is not a good practice in general
        git.checkout("optional_fmt")

    def requirements(self):
        if self.options.with_fmt:
            self.requires("fmt/8.1.1")

    def generate(self):
        tc = CMakeToolchain(self)
        if self.options.with_fmt:
            tc.variables["WITH_FMT"] = True
        tc.generate()

    ...
```

As you can see:

- We declare a new `with_fmt` option with the default value set to `True`
- Based on the value of the `with_fmt` option:
  - We install or not the `fmt/8.1.1` Conan package.
  - We require or not a minimum and a maximum C++ standard as the *fmt* library requires at least C++11 and it will not compile if we try to use a standard above C++14 (just an example, *fmt* can build with more modern standards)
  - We inject the `WITH_FMT` variable with the value `True` to the *CMakeToolchain* so that we can use it in the *CMakeLists.txt* of the **hello** library to add the CMake **fmt::fmt** target conditionally.
- We are cloning another branch of the library. The *optional_fmt* branch contains some changes in the code. Let's see what changed on the CMake side:

Listing 42: **CMakeLists.txt**

```
cmake_minimum_required(VERSION 3.15)
project(hello CXX)

add_library(hello src/hello.cpp)
target_include_directories(hello PUBLIC include)
set_target_properties(hello PROPERTIES PUBLIC_HEADER "include/hello.h")

if (WITH_FMT)
    find_package(fmt)
    target_link_libraries(hello fmt::fmt)
    target_compile_definitions(hello PRIVATE USING_FMT=1)
endif()

install(TARGETS hello)
```

As you can see, we use the `WITH_FMT` we injected in the *CMakeToolchain*. Depending on the value we will try to find the fmt library and link our hello library with it. Also, check that we add the `USING_FMT=1` compile definition that we use in the source code depending on whether we choose to add support for `fmt` or not.

Listing 43: **hello.cpp**

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

#if USING_FMT == 1
#include <fmt/color.h>
#endif

void hello(){
    #if USING_FMT == 1
        #ifdef NDEBUG
        fmt::print(fg(fmt::color::crimson) | fmt::emphasis::bold, "hello/1.0: Hello␣
↪World Release! (with color!)\n");
        #else
        fmt::print(fg(fmt::color::crimson) | fmt::emphasis::bold, "hello/1.0: Hello␣
↪World Debug! (with color!)\n");
        #endif
    #else
        #ifdef NDEBUG
        std::cout << "hello/1.0: Hello World Release! (without color)" << std::endl;
        #else
        std::cout << "hello/1.0: Hello World Debug! (without color)" << std::endl;
        #endif
    #endif
}
```

Let's build the package from sources first using `with_fmt=True` and then `with_fmt=False`. When *test_package* runs it will show different messages depending on the value of the option.

```
$ conan create . --build=missing -o with_fmt=True
-------- Exporting the recipe ----------
...

-------- Testing the package: Running test() ----------
hello/1.0 (test package): Running test()
hello/1.0 (test package): RUN: ./example
hello/1.0: Hello World Release! (with color!)

$ conan create . --build=missing -o with_fmt=False
-------- Exporting the recipe ----------
...

-------- Testing the package: Running test() ----------
hello/1.0 (test package): Running test()
hello/1.0 (test package): RUN: ./example
hello/1.0: Hello World Release! (without color)
```

This is just a simple example of how to use the `generate()` method to customize the toolchain based on the value of one option, but there are lots of other things that you could do in the `generate()` method like:

- Create a complete custom toolchain based on your needs to use in your build.
- **Access to certain information about the package dependencies, like:**
  - The configuration accessing the defined *conf_info*.
  - Accessing the dependencies options.

  - Import files from dependencies using the *copy tool*. You could also import the files create manifests for the package, collecting all dependencies versions and licenses.

- Use the *Environment tools* to generate information for the system environment.
- Adding custom configurations besides *Release* and *Debug*, taking into account the settings, like *ReleaseShared* or *DebugShared*.

Read more

- Use the `generate()` method to import files from dependencies.
- More based on the examples mentioned above . . .

See also:

- *generate() method reference*

4.2.5 Configure settings and options in recipes

We already explained *Conan settings and options* and how to use them to build your projects for different configurations like Debug, Release, with static or shared libraries, etc. In this section, we explain how to configure these settings and options in the case that one of them does not apply to a Conan package. We will introduce briefly how Conan models binary compatibility and how that relates to options and settings.

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

```
$ git clone https://github.com/conan-io/examples2.git
$ cd examples2/tutorial/creating_packages/configure_options_settings
```

You will notice some changes in the **conanfile.py** file from the previous recipe. Let's check the relevant parts:

```
class helloRecipe(ConanFile):
    name = "hello"
    version = "1.0"

    ...
    options = {"shared": [True, False],
               "fPIC": [True, False],
               "with_fmt": [True, False]}

    default_options = {"shared": False,
                       "fPIC": True,
                       "with_fmt": True}
    ...

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

    def configure(self):
        if self.options.shared:
            # If os=Windows, fPIC will have been removed in config_options()
            # use rm_safe to avoid double delete errors
            self.options.rm_safe("fPIC")
    ...
```

You can see that we added a *configure() method* to the recipe. Let's explain what's the objective of this method and how it's different from the `config_options()` method we already had defined in the recipe:

- `configure()`: use this method to configure which options or settings of the recipe are available. For example, in this case, we **delete the fPIC option**, because it should only be **True** if we are building the library as shared (in fact, some build systems will add this flag automatically when building a shared library).
- `config_options()`: This method is used to **constraint** the available options in a package **before they take a value**. If a value is assigned to a setting or option that is deleted inside this method, Conan will raise an error. In this case we are **deleting the fPIC option** in Windows because that option does not exist for that operating system. Note that this method is executed before the `configure()` method.

Be aware that deleting an option in the `config_options()` or in the `configure()` has not the same result. Deleting it in the `config_options()` **is like if we never declared it in the recipe** and it will raise an exception saying that the option does not exist. Nevertheless, if we delete it in the `configure()` method we can pass the option but it will have no effect. For example, if you try to pass a value to the `fPIC` option in Windows, Conan will raise an error warning that the option does not exist:

Listing 44: Windows

```
$ conan create . --build=missing -o fPIC=True
...
-------- Computing dependency graph --------
ERROR: option 'fPIC' doesn't exist
Possible options are ['shared', 'with_fmt']
```

As you have noticed, the `configure()` and `config_options()` methods **delete an option** if certain conditions meet. Let's explain why we are doing this and the implications of removing that option. It is related to how Conan identifies packages that are binary compatible with the configuration set in the profile. In the next section, we introduce the concept of the **Conan package ID**.

Conan packages binary compatibility: the package ID

We used Conan in previous examples to build for different configurations like *Debug* and *Release*. Each time you create the package for one of those configurations, Conan will build a new binary. Each of them is related to a **generated hash** called **the package ID**. The package ID is just a way to convert a set of settings, options and information about the requirements of the package to a unique identifier.

Let's build our package for *Release* and *Debug* configurations and check the generated binaries package IDs.

```
$ conan create . --build=missing -s build_type=Release -tf=None # -tf=None will skip␣
↪buildiing the test_package
...
[ 50%] Building CXX object CMakeFiles/hello.dir/src/hello.cpp.o
[100%] Linking CXX static library libhello.a
[100%] Built target hello
hello/1.0: Package '738feca714b7251063cc51448da0cf4811424e7c' built
hello/1.0: Build folder /Users/user/.conan2/p/tmp/7fe7f5af0ef27552/b/build/Release
hello/1.0: Generated conaninfo.txt
hello/1.0: Generating the package
hello/1.0: Temporary package folder /Users/user/.conan2/p/tmp/7fe7f5af0ef27552/p
hello/1.0: Calling package()
hello/1.0: CMake command: cmake --install "/Users/user/.conan2/p/tmp/7fe7f5af0ef27552/
↪b/build/Release" --prefix "/Users/user/.conan2/p/tmp/7fe7f5af0ef27552/p"
hello/1.0: RUN: cmake --install "/Users/user/.conan2/p/tmp/7fe7f5af0ef27552/b/build/
↪Release" --prefix "/Users/user/.conan2/p/tmp/7fe7f5af0ef27552/p"
-- Install configuration: "Release"
```

(continues on next page)

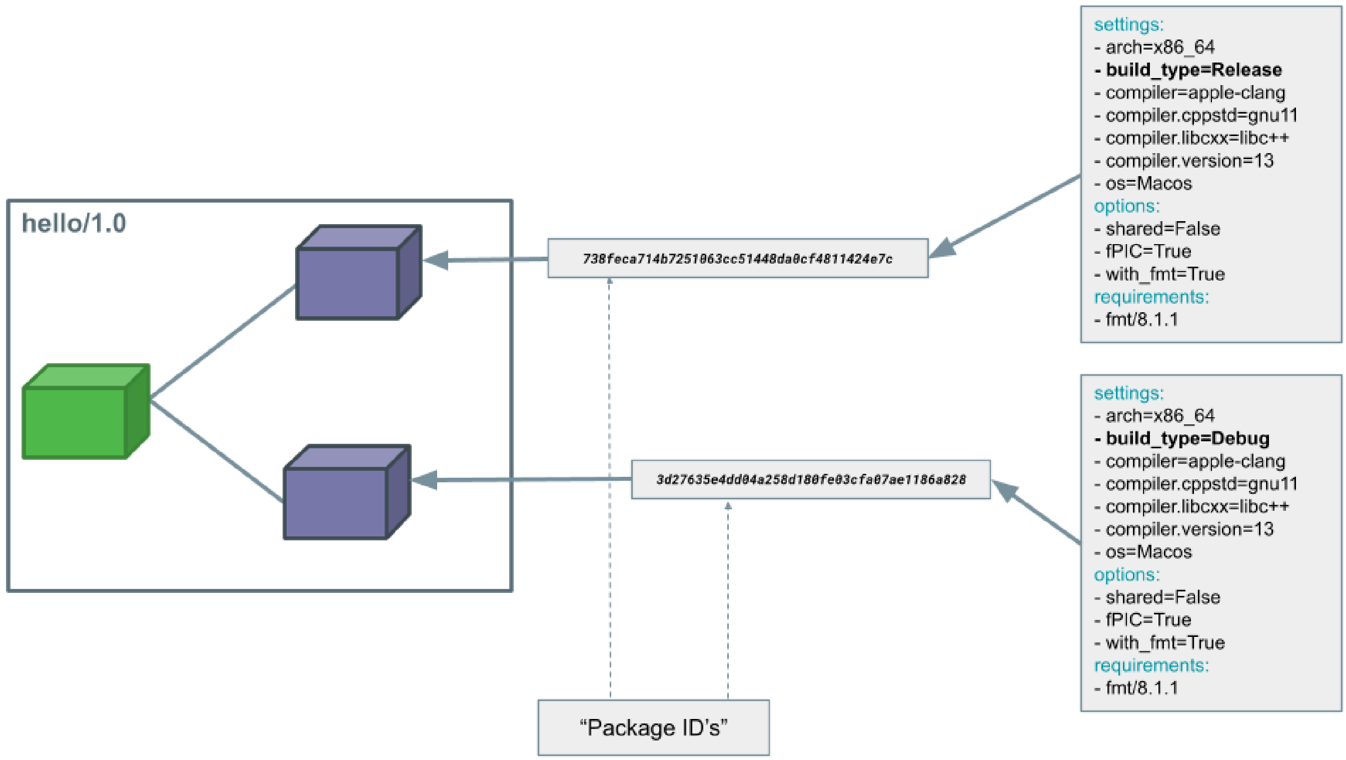
(continued from previous page)

```
-- Installing: /Users/user/.conan2/p/tmp/7fe7f5af0ef27552/p/lib/libhello.a
-- Installing: /Users/user/.conan2/p/tmp/7fe7f5af0ef27552/p/include/hello.h
hello/1.0 package(): Packaged 1 '.h' file: hello.h
hello/1.0 package(): Packaged 1 '.a' file: libhello.a
hello/1.0: Package '738feca714b7251063cc51448da0cf4811424e7c' created
hello/1.0: Created package revision 3bd9faedc711cbb4fdf10b295268246e
hello/1.0: Full package reference: hello/1.0
↪#e6b11fb0cb64e3777f8d62f4543cd6b3:738feca714b7251063cc51448da0cf4811424e7c
↪#3bd9faedc711cbb4fdf10b295268246e
hello/1.0: Package folder /Users/user/.conan2/p/5c497cbb5421cbda/p

$ conan create . --build=missing -s build_type=Debug -tf=None # -tf=None will skip␣
↪buildiing the test_package
...
[ 50%] Building CXX object CMakeFiles/hello.dir/src/hello.cpp.o
[100%] Linking CXX static library libhello.a
[100%] Built target hello
hello/1.0: Package '3d27635e4dd04a258d180fe03cfa07ae1186a828' built
hello/1.0: Build folder /Users/user/.conan2/p/tmp/19a2e552db727a2b/b/build/Debug
hello/1.0: Generated conaninfo.txt
hello/1.0: Generating the package
hello/1.0: Temporary package folder /Users/user/.conan2/p/tmp/19a2e552db727a2b/p
hello/1.0: Calling package()
hello/1.0: CMake command: cmake --install "/Users/user/.conan2/p/tmp/19a2e552db727a2b/
↪b/build/Debug" --prefix "/Users/user/.conan2/p/tmp/19a2e552db727a2b/p"
hello/1.0: RUN: cmake --install "/Users/user/.conan2/p/tmp/19a2e552db727a2b/b/build/
↪Debug" --prefix "/Users/user/.conan2/p/tmp/19a2e552db727a2b/p"
-- Install configuration: "Debug"
-- Installing: /Users/user/.conan2/p/tmp/19a2e552db727a2b/p/lib/libhello.a
-- Installing: /Users/user/.conan2/p/tmp/19a2e552db727a2b/p/include/hello.h
hello/1.0 package(): Packaged 1 '.h' file: hello.h
hello/1.0 package(): Packaged 1 '.a' file: libhello.a
hello/1.0: Package '3d27635e4dd04a258d180fe03cfa07ae1186a828' created
hello/1.0: Created package revision 67b887a0805c2a535b58be404529c1fe
hello/1.0: Full package reference: hello/1.0
↪#e6b11fb0cb64e3777f8d62f4543cd6b3:3d27635e4dd04a258d180fe03cfa07ae1186a828
↪#67b887a0805c2a535b58be404529c1fe
hello/1.0: Package folder /Users/user/.conan2/p/c7796386fcad5369/p
```

As you can see Conan generated two package IDs:

- Package *738feca714b7251063cc51448da0cf4811424e7c* for Release
- Package *3d27635e4dd04a258d180fe03cfa07ae1186a828* for Debug

These two package IDs are calculated by taking the **set of settings, options and some information about the requirements** (we will explain this later in the documentation) and **calculating a hash** with them. So, for example, in this case, they are the result of the information depicted in the diagram below.



Those package IDs are different because the **build_type** is different. Now, when you want to install a package, Conan will:

- Collect the settings and options applied, along with some information about the requirements and calculate the hash for the corresponding package ID.
- If that package ID matches one of the packages stored in the local Conan cache Conan will use that. If not, and we have any Conan remote configured, it will search for a package with that package ID in the remotes.
- If that calculated package ID does not exist in the local cache and remotes, Conan will fail with a "missing binary" error message, or will try to build that package from sources (this depends on the value of the `--build` argument). This build will generate a new package ID in the local cache.

This steps are simplified, there is far more to package ID calculation than what we explain here, recipes themselves can even adjust their package ID calculations, we can have different recipe and package revisions besides package IDs and there's also a built-in mechanism in Conan that can be configured to declare that some packages with a certain package ID are compatible with other.

Maybe you have now the intuition of why we delete settings or options in Conan recipes. If you do that, those values will not be added to the computation of the package ID, so even if you define them, the resulting package ID will be the same. You can check this behaviour, for example with the fPIC option that is deleted when we build with the option `shared=True`. Regardless of the value you pass for the fPIC option the generated package ID will be the same for the **hello/1.0** binary:

```
$ conan conan create . --build=missing -o shared=True -o fPIC=True -tf=None
...
hello/1.0 package(): Packaged 1 '.h' file: hello.h
hello/1.0 package(): Packaged 1 '.dylib' file: libhello.dylib
hello/1.0: Package '2a899fd0da3125064bf9328b8db681cd82899d56' created
hello/1.0: Created package revision f0d1385f4f90ae465341c15740552d7e
hello/1.0: Full package reference: hello/1.0
↪#e6b11fb0cb64e3777f8d62f4543cd6b3:2a899fd0da3125064bf9328b8db681cd82899d56
↪#f0d1385f4f90ae465341c15740552d7e
```

(continues on next page)

(continued from previous page)

```
hello/1.0: Package folder /Users/user/.conan2/p/8a55286c6595f662/p

$ conan conan create . --build=missing -o shared=True -o fPIC=False -tf=None
...
-------- Computing dependency graph --------
Graph root
    virtual
Requirements
    fmt/8.1.1#601209640bd378c906638a8de90070f7 - Cache
    hello/1.0#e6b11fb0cb64e3777f8d62f4543cd6b3 - Cache

-------- Computing necessary packages --------
Requirements
    fmt/8.1.1
↪#601209640bd378c906638a8de90070f7:d1b3f3666400710fec06446a697f9eeddd1235aa
↪#24a2edf207deeed4151bd87bca4af51c - Skip
    hello/1.0
↪#e6b11fb0cb64e3777f8d62f4543cd6b3:2a899fd0da3125064bf9328b8db681cd82899d56
↪#f0d1385f4f90ae465341c15740552d7e - Cache

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

-------- Installing (downloading, building) binaries... --------
hello/1.0: Already installed!
```

As you can see, the first run created the `2a899fd0da3125064bf9328b8db681cd82899d56` package, and the second one, regardless of the different value of the fPIC option, said we already had the `2a899fd0da3125064bf9328b8db681cd82899d56` package installed.

C libraries

There are other typical cases where you want to delete certain settings. Imagine that you are packaging a C library. When you build this library, there are settings like the compiler C++ standard (`settings.compiler.cppstd`) or the standard library used (`self.settings.compiler.libcxx`) that won't affect the resulting binary at all. Then it does no make sense that they affect to the package ID computation, so a typical pattern is to delete them in the `configure()` method:

```python
def configure(self):
    del self.settings.compiler.cppstd
    del self.settings.compiler.libcxx
```

Please, note that deleting these settings in the `configure()` method will modify the package ID calculation but will also affect how the toolchain, and the build system integrations work because the C++ settings do not exist.

Header-only libraries

A similar case happens with packages that package *header-only libraries*. In that case, there's no binary code we need to link with, but just some header files to add to our project. In this cases the package ID of the Conan package should not be affected by settings or options. For that case, there's a simplified way of declaring that the generated package ID should not take into account settings, options or any information from the requirement which is using the `self.info.clear()` method inside another recipe method called `package_id()`:

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

We will explain the `package_id()` method later and explain how you can customize the way the package ID for the package is calculated. You can also check the *Conanfile's methods reference* if you want to know how this method works in more detail.

Read more

- *Header-only packages*.
- Check the binary compatibility *compatibility.py extension*.
- Conan *package types*.
- *Setting package_id_mode for requirements*.

4.2.6 Build packages: the build() method

We already used a Conan recipe that has a *build() method* and learned how to use that to invoke a build system and build our packages. In this tutorial, we will modify that method and explain how you can use it to do things like:

- Building and running tests
- Conditional patching of the source code
- Select the build system you want to use conditionally

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

```
$ git clone https://github.com/conan-io/examples2.git
$ cd examples2/tutorial/creating_packages/build_method
```

Build and run tests for your project

You will notice some changes in the **conanfile.py** file from the previous recipe. Let's check the relevant parts:

Changes introduced in the recipe

Listing 45: *conanfile.py*

```
class helloRecipe(ConanFile):
    name = "hello"
    version = "1.0"

    ...

    def source(self):
        git = Git(self)
        git.clone(url="https://github.com/conan-io/libhello.git", target=".")
        # Please, be aware that using the head of the branch instead of an immutable␣
↪tag
        # or commit is not a good practice in general
        git.checkout("with_tests")
```

(continues on next page)

(continued from previous page)

```
...

def requirements(self):
    if self.options.with_fmt:
        self.requires("fmt/8.1.1")
    self.test_requires("gtest/1.11.0")

...

def generate(self):
    tc = CMakeToolchain(self)
    if self.options.with_fmt:
        tc.variables["WITH_FMT"] = True
    tc.generate()

def build(self):
    cmake = CMake(self)
    cmake.configure()
    cmake.build()
    if not self.conf.get("tools.build:skip_test", default=False):
        test_folder = os.path.join("tests")
        if self.settings.os == "Windows":
            test_folder = os.path.join("tests", str(self.settings.build_type))
        self.run(os.path.join(test_folder, "test_hello"))

...
```

- We added the *gtest/1.11.0* requirement to the recipe as a `test_requires()`. It's a type of requirement intended for testing libraries like **Catch2** or **gtest**.
- We use the `tools.build:skip_test` configuration (`False` by default), to tell CMake whether to build and run the tests or not. A couple of things to bear in mind:
  - If we set the `tools.build:skip_test` configuration to `True` Conan will automatically inject the `BUILD_TESTING` variable to CMake set to `OFF`. You will see in the next section that we are using this variable in our *CMakeLists.txt* to decide wether to build the tests or not.
  - We use the `tools.build:skip_test` configuration in the `build()` method, after building the package and tests, to decide if we want to run the tests or not.
  - In this case we are using **gtest** for testing and we have to add the check if the build method to run the tests or not, but this configuration also affects the execution of `CMake.test()` if you are using CTest and `Meson.test()` for Meson.

Changes introduced in the library sources

First, please note that we are using another branch from the **libhello** library. This branch has two novelties on the library side:

- We added a new function called `compose_message()` to the library sources so we can add some unit tests over this function. This function is just creating an output message based on the arguments passed.
- As we mentioned in the previous section the CMakeLists.txt for the library uses the `BUILD_TESTING` CMake variable that conditionally adds the *tests* directory.

Listing 46: *CMakeLists.txt*

```
cmake_minimum_required(VERSION 3.15)
project(hello CXX)

...

if (NOT BUILD_TESTING STREQUAL OFF)
    add_subdirectory(tests)
endif()

...
```

The `BUILD_TESTING` CMake variable is declared and set to `OFF` by Conan (if not already defined) whenever the `tools.build:skip_test` configuration is set to value `True`. This variable is typically declared by CMake when you use CTest but using the `tools.build:skip_test` configuration you can use it in your *CMakeListst.txt* even if you are using another testing framework.

- We have a CMakeLists.txt in the *tests* folder using googletest for testing.

Listing 47: *tests/CMakeLists.txt*

```
cmake_minimum_required(VERSION 3.15)
project(PackageTest CXX)

find_package(GTest REQUIRED CONFIG)

add_executable(test_hello test.cpp)
target_link_libraries(test_hello GTest::gtest GTest::gtest_main hello)
```

With basic tests on the functionality of the `compose_message()` function:

Listing 48: *tests/test.cpp*

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

namespace {
    TEST(HelloTest, ComposeMessages) {
    EXPECT_EQ(std::string("hello/1.0: Hello World Release! (with color!)\n"), compose_
↪message("Release", "with color!"));
    ...
    }
}
```

Now that we have gone through all the changes in the code, let's try them out:

```
$ conan create . --build=missing -tf=None
...
[ 25%] Building CXX object CMakeFiles/hello.dir/src/hello.cpp.o
[ 50%] Linking CXX static library libhello.a
[ 50%] Built target hello
[ 75%] Building CXX object tests/CMakeFiles/test_hello.dir/test.cpp.o
[100%] Linking CXX executable test_hello
[100%] Built target test_hello
hello/1.0: RUN: ./tests/test_hello
Capturing current environment in /Users/user/.conan2/p/tmp/c51d80ef47661865/b/build/
↪generators/deactivate_conanbuildenv-release-x86_64.sh
```

(continues on next page)

(continued from previous page)

```
Configuring environment variables
Running main() from /Users/user/.conan2/p/tmp/3ad4c6873a47059c/b/googletest/src/gtest_
↪main.cc
[==========] Running 1 test from 1 test suite.
[----------] Global test environment set-up.
[----------] 1 test from HelloTest
[ RUN      ] HelloTest.ComposeMessages
[       OK ] HelloTest.ComposeMessages (0 ms)
[----------] 1 test from HelloTest (0 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test suite ran. (0 ms total)
[  PASSED  ] 1 test.
hello/1.0: Package '82b6c0c858e739929f74f59c25c187b927d514f3' built
...
```

As you can see, the tests were built and run. Let's use now the `tools.build:skip_test` configuration in the command line to skip the test building and running:

```
$ conan create . -c tools.build:skip_test=True -tf=None
...
[ 50%] Building CXX object CMakeFiles/hello.dir/src/hello.cpp.o
[100%] Linking CXX static library libhello.a
[100%] Built target hello
hello/1.0: Package '82b6c0c858e739929f74f59c25c187b927d514f3' built
...
```

You can see now that only the library target was built and that no tests were built or run.

Conditionally patching the source code

If you need to patch the source code the recommended approach is to do that in the `source()` method. Sometimes, if that patch depends on settings or options, you have to use the `build()` method to apply patches to the source code before launching the build. There are *several ways to do this* in Conan. One of them would be using the *replace_in_file* tool:

```
import os
from conan import ConanFile
from conan.tools.files import replace_in_file


class helloRecipe(ConanFile):
    name = "hello"
    version = "1.0"

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

    def build(self):
        replace_in_file(self, os.path.join(self.source_folder, "src", "hello.cpp"),
                        "Hello World",
                        "Hello {} Friends".format("Shared" if self.options.shared␣
↪else "Static"))
```

Please, note that patching in `build()` should avoided if possible and only be done for very particular cases as it will make more difficult to develop your packages locally (we will explain more about this in the local developement flow section later <MISSING REFERENCE>)

Conditionally select your build system

It's not uncommon that some packages need one build system or another depending on the platform we are building. For example, the *hello* library could build in Windows using CMake and in Linux and MacOS using Autotools. This can be easily handled in the `build()` method like this:

```
...

class helloRecipe(ConanFile):
    name = "hello"
    version = "1.0"

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

    ...

    def generate(self):
        if self.settings.os == "Windows":
            tc = CMakeToolchain(self)
            tc.generate()
            deps = CMakeDeps(self)
            deps.generate()
        else:
            tc = AutotoolsToolchain(self)
            tc.generate()
            deps = PkgConfigDeps(self)
            deps.generate()

    ...

    def build(self):
        if self.settings.os == "Windows":
            cmake = CMake(self)
            cmake.configure()
            cmake.build()
        else:
            autotools = Autotools(self)
            autotools.autoreconf()
            autotools.configure()
            autotools.make()

    ...
```

Read more

- *Patching sources*
- ...

4.2.7 Package files: the package() method

We already used the `package()` method in our *hello* package to invoke CMake's install step. In this tutorial, we will explain the use of the *CMake.install()* in more detail and also how to modify this method to do things like:

- Using *conan.tools.files* utilities to copy the generated artifacts from the build folder to the package folder
- Copying package licenses
- Manage how to package symlinks

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

```
$ git clone https://github.com/conan-io/examples2.git
$ cd examples2/tutorial/creating_packages/package_method
```

Using CMake install step in the package() method

This is the simplest choice when you have already defined in your *CMakeLists.txt* the functionality of extracting the artifacts (headers, libraries, binaries) from the build and source folder to a predetermined place and maybe do some post-processing of those artifacts. This will work without changes in your *CMakeLists.txt* because Conan will set the `CMAKE_INSTALL_PREFIX` CMake variable to point to the recipe's *package_folder* attribute. Then, just calling *install()* in the *CMakeLists.txt* over the created target is enough for Conan to move the built artifacts to the correct location in the Conan local cache.

Listing 49: *CMakeLists.txt*

```
cmake_minimum_required(VERSION 3.15)
project(hello CXX)

add_library(hello src/hello.cpp)
target_include_directories(hello PUBLIC include)
set_target_properties(hello PROPERTIES PUBLIC_HEADER "include/hello.h")

...

install(TARGETS hello)
```

Listing 50: *conanfile.py*

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

Let's build our package again and pay attention to the lines regarding the packaging of files in the Conan local cache:

```
$ conan create . --build=missing -tf=None
...
hello/1.0: Build folder /Users/user/.conan2/p/tmp/b5857f2e70d1b2fd/b/build/Release
hello/1.0: Generated conaninfo.txt
hello/1.0: Generating the package
hello/1.0: Temporary package folder /Users/user/.conan2/p/tmp/b5857f2e70d1b2fd/p
hello/1.0: Calling package()
hello/1.0: CMake command: cmake --install "/Users/user/.conan2/p/tmp/b5857f2e70d1b2fd/
↪b/build/Release" --prefix "/Users/user/.conan2/p/tmp/b5857f2e70d1b2fd/p"
hello/1.0: RUN: cmake --install "/Users/user/.conan2/p/tmp/b5857f2e70d1b2fd/b/build/
↪Release" --prefix "/Users/user/.conan2/p/tmp/b5857f2e70d1b2fd/p"
```

(continues on next page)

(continued from previous page)

```
-- Install configuration: "Release"
-- Installing: /Users/user/.conan2/p/tmp/b5857f2e70d1b2fd/p/lib/libhello.a
-- Installing: /Users/user/.conan2/p/tmp/b5857f2e70d1b2fd/p/include/hello.h
hello/1.0 package(): Packaged 1 '.h' file: hello.h
hello/1.0 package(): Packaged 1 '.a' file: libhello.a
hello/1.0: Package 'fd7c4113dad406f7d8211b3470c16627b54ff3af' created
hello/1.0: Created package revision bf7f5b9a3bb2c957742be4be216dfcbb
hello/1.0: Full package reference: hello/1.0
↪#25e0b5c00ae41ef9fbfbbb1e5ac86e1e:fd7c4113dad406f7d8211b3470c16627b54ff3af
↪#bf7f5b9a3bb2c957742be4be216dfcbb
hello/1.0: Package folder /Users/user/.conan2/p/47b4c4c61c8616e5/p
```

As you can see both the *include* and *library* files were copied to the package folder after calling to the `cmake.install()` method.

Use conan.tools.files.copy() in the package() method and packaging licenses

For the cases that you don't want to rely on CMake's install functionality or that you are using another build-system, Conan provides the tools to copy the selected files to the *package_folder*. In this case, you can use the *tools.files.copy* function to make that copy. We can replace the previous `cmake.install()` step with a custom copy of the files and the result would be the same.

Note that we are also packaging the `LICENSE` file from the library sources in the *licenses* folder. This is a common pattern in Conan packages and could also be added to the previous example using `cmake.install()` as the *CMakeLists.txt* will not copy this file to the *package folder*.

Listing 51: *conanfile.py*

```
def package(self):
    copy(self, "LICENSE", src=self.source_folder, dst=os.path.join(self.package_
↪folder, "licenses"))
    copy(self, pattern="*.h", src=os.path.join(self.source_folder, "include"), dst=os.
↪path.join(self.package_folder, "include"))
    copy(self, pattern="*.a", src=self.build_folder, dst=os.path.join(self.package_
↪folder, "lib"), keep_path=False)
    copy(self, pattern="*.so", src=self.build_folder, dst=os.path.join(self.package_
↪folder, "lib"), keep_path=False)
    copy(self, pattern="*.lib", src=self.build_folder, dst=os.path.join(self.package_
↪folder, "lib"), keep_path=False)
    copy(self, pattern="*.dll", src=self.build_folder, dst=os.path.join(self.package_
↪folder, "bin"), keep_path=False)
    copy(self, pattern="*.dylib", src=self.build_folder, dst=os.path.join(self.
↪package_folder, "lib"), keep_path=False)
```

Let's build our package one more time and pay attention to the lines regarding the packaging of files in the Conan local cache:

```
$ conan create . --build=missing -tf=None
...
hello/1.0: Build folder /Users/user/.conan2/p/tmp/222db0532bba7cbc/b/build/Release
hello/1.0: Generated conaninfo.txt
hello/1.0: Generating the package
hello/1.0: Temporary package folder /Users/user/.conan2/p/tmp/222db0532bba7cbc/p
hello/1.0: Calling package()
hello/1.0: Copied 1 file: LICENSE
```

(continues on next page)

(continued from previous page)

```
hello/1.0: Copied 1 '.h' file: hello.h
hello/1.0: Copied 1 '.a' file: libhello.a
hello/1.0 package(): Packaged 1 file: LICENSE
hello/1.0 package(): Packaged 1 '.h' file: hello.h
hello/1.0 package(): Packaged 1 '.a' file: libhello.a
hello/1.0: Package 'fd7c4113dad406f7d8211b3470c16627b54ff3af' created
hello/1.0: Created package revision 50f91e204d09b64b24b29df3b87a2f3a
hello/1.0: Full package reference: hello/1.0
↪#96ed9fb1f78bc96708b1abf4841523b0:fd7c4113dad406f7d8211b3470c16627b54ff3af
↪#50f91e204d09b64b24b29df3b87a2f3a
hello/1.0: Package folder /Users/user/.conan2/p/21ec37b931782de8/p
```

Check how the *include* and *library* files are packaged. The LICENSE file is also copied as we explained above.

Managing symlinks in the package() method

Another thing you can do in the package method is managing how to package symlinks. Conan won't manipulate symlinks by default, so we provide several *tools* to convert absolute symlinks to relative ones and removing external or broken symlinks.

Imagine that some of the files packaged in the latest example were symlinks that point to an absolute location inside the Conan cache. Then, calling to `conan.tools.files.symlinks.absolute_to_relative_symlinks()` would convert those absolute links into relative paths and make the package relocatable.

Listing 52: *conanfile.py*

```python
from conan.tools.files.symlinks import absolute_to_relative_symlinks

def package(self):
    copy(self, "LICENSE", src=self.source_folder, dst=os.path.join(self.package_
↪folder, "licenses"))
    copy(self, pattern="*.h", src=os.path.join(self.source_folder, "include"), dst=os.
↪path.join(self.package_folder, "include"))
    copy(self, pattern="*.a", src=self.build_folder, dst=os.path.join(self.package_
↪folder, "lib"), keep_path=False)
    ...

    absolute_to_relative_symlinks(self, self.package_folder)
```

Read more

- ...

See also:

- *package() method reference*

4.2.8 Define information for consumers: the package_info() method

In the previous tutorial section, we explained how to store the headers and binaries of a library in a Conan package using the *package method*. Consumers that depend on that package will reuse those files, but we have to provide some additional information so that Conan can pass that to the build system and consumers can use the package.

For instance, in our example, we are building a static library named *hello* that will result in a *libhello.a* file in Linux and macOS or a *hello.lib* file in Windows. Also, we are packaging a header file *hello.h* with the declaration of the library functions. The Conan package ends up with the following structure in the Conan local cache:

```
.
├── include
│   └── hello.h
└── lib
    └── libhello.a
```

Then, consumers that want to link against this library will need some information:

- The location of the *include* folder in the Conan local cache to search for the *hello.h* file.
- The name of the library file to link against it (*libhello.a* or *hello.lib*)
- The location of the *lib* folder in the Conan local cache to search for the library file.

Conan provides an abstraction over all the information consumers may need in the *cpp_info* attribute of the ConanFile. The information for this attribute must be set in the *package_info() method*. Let's have a look at the `package_info()` method of our *hello/1.0* Conan package:

Listing 53: *conanfile.py*

```
...

class helloRecipe(ConanFile):
    name = "hello"
    version = "1.0"

    ...

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

We can see a couple of things:

- We are adding a *hello* library to the `libs` property of the `cpp_info` to tell consumers that they should link the libraries from that list.
- We are **not adding** information about the *lib* or *include* folders where the library and headers files are packaged. The `cpp_info` object provides the `.includedirs` and `.libdirs` properties to define those locations but Conan sets their value as `lib` and `include` by default so it's not needed to add those in this case. If you were copying the package files to a different location then you have to set those explicitly. The declaration of the `package_info` method in our Conan package would be equivalent to this one:

Listing 54: *conanfile.py*

```
...

class helloRecipe(ConanFile):
    name = "hello"
    version = "1.0"

    ...

    def package_info(self):
        self.cpp_info.libs = ["hello"]
        # conan sets libdirs = ["lib"] and includedirs = ["include"] by default
```

(continues on next page)

(continued from previous page)

```
        self.cpp_info.libdirs = ["lib"]
        self.cpp_info.includedirs = ["include"]
```

Setting information in the package_info() method

Besides what we explained above about the information you can set in the `package_info()` method, there are some typical use cases:

- Define information for consumers depending on settings or options
- Customizing certain information that generators provide to consumers, like the target names for CMake or the generated files names for pkg-config for example
- Propagating configuration values to consumers
- Propagating environment information to consumers
- Define components for Conan packages that provide multiple libraries

Let's see some of those in action. First, clone the project sources if you haven't done so yet. You can find them in the examples2.0 repository on GitHub:

```
$ git clone https://github.com/conan-io/examples2.git
$ cd examples2/tutorial/creating_packages/package_information
```

Define information for consumers depending on settings or options

For this section of the tutorial we introduced some changes in the library and recipe. Let's check the relevant parts:

Changes introduced in the library sources

First, please note that we are using another branch from the **libhello** library. Let's check the library's *CMakeLists.txt*:

Listing 55: *CMakeLists.txt*

```
cmake_minimum_required(VERSION 3.15)
project(hello CXX)

...

add_library(hello src/hello.cpp)

if (BUILD_SHARED_LIBS)
    set_target_properties(hello PROPERTIES OUTPUT_NAME hello-shared)
else()
    set_target_properties(hello PROPERTIES OUTPUT_NAME hello-static)
endif()

...
```

As you can see, we are setting the output name for the library depending on whether we are building the library as static (*hello-static*) or as shared (*hello-shared*). Now let's see how to translate these changes to the Conan recipe.

Changes introduced in the recipe

To update our recipe according to the changes in the library's *CMakeLists.txt* we have to conditionally set the library name depending on the `self.options.shared` option in the `package_info()` method:

Listing 56: *conanfile.py*

```
class helloRecipe(ConanFile):
    ...

    def source(self):
        git = Git(self)
        git.clone(url="https://github.com/conan-io/libhello.git", target=".")
        # Please, be aware that using the head of the branch instead of an immutable␣
↪tag
        # or commit is not a good practice in general
        git.checkout("package_info")

    ...

    def package_info(self):
        if self.options.shared:
            self.cpp_info.libs = ["hello-shared"]
        else:
            self.cpp_info.libs = ["hello-static"]
```

Now, let's create the Conan package with `shared=False` (that's the default so no need to set it explicitly) and check that we are packaging the correct library (*libhello-static.a* or *hello-static.lib*) and that we are linking the correct library in the *test_package*.

```
$ conan create . --build=missing
...
-- Install configuration: "Release"
-- Installing: /Users/user/.conan2/p/tmp/a311fcf8a63f3206/p/lib/libhello-static.a
-- Installing: /Users/user/.conan2/p/tmp/a311fcf8a63f3206/p/include/hello.h
hello/1.0 package(): Packaged 1 '.h' file: hello.h
hello/1.0 package(): Packaged 1 '.a' file: libhello-static.a
hello/1.0: Package 'fd7c4113dad406f7d8211b3470c16627b54ff3af' created
...
-- Build files have been written to: /Users/user/.conan2/p/tmp/a311fcf8a63f3206/b/
↪build/Release
hello/1.0: CMake command: cmake --build "/Users/user/.conan2/p/tmp/a311fcf8a63f3206/b/
↪build/Release" -- -j16
hello/1.0: RUN: cmake --build "/Users/user/.conan2/p/tmp/a311fcf8a63f3206/b/build/
↪Release" -- -j16
[ 25%] Building CXX object CMakeFiles/hello.dir/src/hello.cpp.o
[ 50%] Linking CXX static library libhello-static.a
[ 50%] Built target hello
[ 75%] Building CXX object tests/CMakeFiles/test_hello.dir/test.cpp.o
[100%] Linking CXX executable test_hello
[100%] Built target test_hello
hello/1.0: RUN: tests/test_hello
...
[ 50%] Building CXX object CMakeFiles/example.dir/src/example.cpp.o
[100%] Linking CXX executable example
[100%] Built target example
```

(continues on next page)

(continued from previous page)

```
-------- Testing the package: Running test() --------
hello/1.0 (test package): Running test()
hello/1.0 (test package): RUN: ./example
hello/1.0: Hello World Release! (with color!)
```

As you can see both the tests for the library and the Conan *test_package* linked against the *libhello-static.a* library successfully.

Properties model: setting information for specific generators

The *CppInfo* object provides the `set_property` method to set information specific to each generator. For example, in this tutorial, we use the *CMakeDeps* generator to generate the information that CMake needs to build a project that requires our library. `CMakeDeps`, by default, will set a target name for the library using the same name as the Conan package. If you have a look at that *CMakeLists.txt* from the *test_package*:

Listing 57: test_package *CMakeLists.txt*

```
cmake_minimum_required(VERSION 3.15)
project(PackageTest CXX)

find_package(hello CONFIG REQUIRED)

add_executable(example src/example.cpp)
target_link_libraries(example hello::hello)
```

You can see that we are linking with the target name `hello::hello`. Conan sets this target name by default, but we can change it using the *properties model*. Let's try to change it to the name `hello::myhello`. To do this, we have to set the property `cmake_target_name` in the package_info method of our *hello/1.0* Conan package:

Listing 58: *conanfile.py*

```
class helloRecipe(ConanFile):
    ...

    def package_info(self):
        if self.options.shared:
            self.cpp_info.libs = ["hello-shared"]
        else:
            self.cpp_info.libs = ["hello-static"]

        self.cpp_info.set_property("cmake_target_name", "hello::myhello")
```

Then, change the target name we are using in the *CMakeLists.txt* in the *test_package* folder to `hello::myhello`:

Listing 59: test_package *CMakeLists.txt*

```
cmake_minimum_required(VERSION 3.15)
project(PackageTest CXX)
# ...
target_link_libraries(example hello::myhello)
```

And re-create the package:

```
$ conan create . --build=missing
Exporting the recipe
```

(continues on next page)

(continued from previous page)

```
hello/1.0: Exporting package recipe
hello/1.0: Using the exported files summary hash as the recipe revision:␣
↪44d78a68b16b25c5e6d7e8884b8f58b8
hello/1.0: A new conanfile.py version was exported
hello/1.0: Folder: /Users/user/.conan2/p/a8cb81b31dc10d96/e
hello/1.0: Exported revision: 44d78a68b16b25c5e6d7e8884b8f58b8
...
-------- Testing the package: Building --------
hello/1.0 (test package): Calling build()
...
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Conan: Target declared 'hello::myhello'
...
[100%] Linking CXX executable example
[100%] Built target example

-------- Testing the package: Running test() --------
hello/1.0 (test package): Running test()
hello/1.0 (test package): RUN: ./example
hello/1.0: Hello World Release! (with color!)
```

You can see how Conan now declares the `hello::myhello` instead of the default `hello::hello` and the *test_package* builds successfully.

The target name is not the only property you can set in the CMakeDeps generator. For a complete list of properties that affect the CMakeDeps generator behaviour, please check the *reference*.

Propagating environment or configuration information to consumers

You can provide environment information to consumers in the `package_info()`. To do so, you can use the ConanFile's *runenv_info* and *buildenv_info* properties:

- `runenv_info` *Environment* object that defines environment information that consumers that use the package may need when **running**.
- `buildenv_info` *Environment* object that defines environment information that consumers that use the package may need when **building**.

Please note that it's not necessary to add `cpp_info.bindirs` to `PATH` or `cpp_info.libdirs` to `LD_LIBRARY_PATH`, those are automatically added by the *VirtualBuildEnv* and *VirtualRunEnv*.

You can also define configuration values in the `package_info()` so that consumers can use that information. To do this, set the *conf_info* property of the ConanFile.

To know more about this use case, please check the *corresponding example*.

Define components for Conan packages that provide multiple libraries

There are cases in which a Conan package may provide multiple libraries, for these cases you can set the separate information for each of those libraries using the components attribute from the *CppInfo* object.

To know more about this use case, please check the *components example* in the examples section.

Read more

- *Propagating environment and configuration information to consumers example*
- *Define components for Conan packages that provide multiple libraries example*

See also:

- *package_info() reference*

4.2.9 Testing Conan packages

In all the previous sections of the tutorial, we used the *test_package*. It was invoked automatically at the end of the `conan create` command after building our package verifying that the package is created correctly. Let's explain the *test_package* in more detail in this section:

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

```
$ git clone https://github.com/conan-io/examples2.git
$ cd examples2/tutorial/creating_packages/testing_packages
```

Some important notes to have in mind about the *test_package*:

- The *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.

The *test_package* folder for our hello/1.0 Conan package has the following contents:

```
test_package
 ├── CMakeLists.txt
 ├── conanfile.py
 └── src
     └── example.cpp
```

Let's have a look at the different files that are part of the *test_package*. First, *example.cpp* is just a minimal example of how to use the *libhello* library that we are packaging:

Listing 60: *test_package/src/example.cpp*

```cpp
#include "hello.h"

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

Then the *CMakeLists.txt* file to tell CMake how to build the example:

Listing 61: *test_package/src/example.cpp*

```cmake
cmake_minimum_required(VERSION 3.15)
project(PackageTest CXX)

find_package(hello CONFIG REQUIRED)
```

(continues on next page)

(continued from previous page)

```
add_executable(example src/example.cpp)
target_link_libraries(example hello::hello)
```

Finally, the recipe for the *test_package* that consumes the *hello/1.0* Conan package:

Listing 62: *test_package/conanfile.py*

```
import os

from conan import ConanFile
from conan.tools.cmake import CMake, cmake_layout
from conan.tools.build import can_run


class helloTestConan(ConanFile):
    settings = "os", "compiler", "build_type", "arch"
    generators = "CMakeDeps", "CMakeToolchain"

    def requirements(self):
        self.requires(self.tested_reference_str)

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

    def layout(self):
        cmake_layout(self)

    def test(self):
        if can_run(self):
            cmd = os.path.join(self.cpp.build.bindir, "example")
            self.run(cmd, env="conanrun")
```

Let's go through the most relevant parts:

- We add the requirements in the `requirements()` method, but in this case we use the `tested_reference_str` attribute that Conan sets to pass to the test_package. This is a convenience attribute to avoid hardcoding the package name in the test_package so that we can reuse the same test_package for several versions of the same Conan package. In our case, this variable will take the `hello/1.0` value.
- We define a `test()` method. This method will only be invoked in the *test_package* recipes. It executes immediately after `build()` is called, and it's meant to run some executable or tests on binaries to prove the package is correctly created. A couple of comments about the contents of our `test()` method:
  - We are using the *conan.tools.build.cross_building* tool to check if we can run the built executable in our platform. This tool will return the value of the `tools.build.cross_building:can_run` in case it's set. Otherwise it will return if we are cross-building or not. It's an useful feature for the case your architecture can run more than one target. For instance, Mac M1 machines can run both *armv8* and *x86_64*.
  - We run the example binary, that was generated in the `self.cpp.build.bindir` folder using the environment information that Conan put in the run environment. Conan will then invoke a launcher containing the runtime environment information, anything that is necessary for the environment to run the compiled executables and applications.

Now that we have gone through all the important bits of the code, let's try our *test_package*. Although we already learned that the *test_package* is invoked when we call to `conan create`, you can also just create the *test_package* if you have already created the `hello/1.0` package in the Conan cache. This is done with the *conan test* command:

```
$ conan test test_package hello/1.0

...

-------- test_package: Computing necessary packages --------
Requirements
    fmt/8.1.1
↪#cd132b054cf999f31bd2fd2424053ddc:ff7a496f48fca9a88dc478962881e015f4a5b98f
↪#1d9bb4c015de50bcb4a338c07229b3bc - Cache
    hello/1.0
↪#25e0b5c00ae41ef9fbfbbb1e5ac86e1e:fd7c4113dad406f7d8211b3470c16627b54ff3af
↪#4ff3fd65a1d37b52436bf62ea6eaac04 - Cache
Test requirements
    gtest/1.11.0
↪#d136b3379fdb29bdfe31404b916b29e1:656efb9d626073d4ffa0dda2cc8178bc408b1bee
↪#ee8cbd2bf32d1c89e553bdd9d5606127 - Skip

...

[ 50%] Building CXX object CMakeFiles/example.dir/src/example.cpp.o
[100%] Linking CXX executable example
[100%] Built target example

-------- Testing the package: Running test() --------
hello/1.0 (test package): Running test()
hello/1.0 (test package): RUN: ./example
hello/1.0: Hello World Release! (with color!)
```

As you can see in the output, our *test_package* builds successfully testing that the *hello/1.0* Conan package can be consumed with no problem.

Read more

- Test *tool_requires* packages
- . . .

4.2.10 Other types of packages

In the previous sections, we saw how to create a new recipe for a classic C++ library but there are other types of packages rather than libraries.

In this section, we are going to review how to create a recipe for `header-only` libraries, how to package already `built libraries`, and how to create recipes for `tool requires` an `applications`.

Header-only packages

In this section, we are going to learn how to create a recipe for a header-only library.

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

```
$ git clone https://github.com/conan-io/examples2.git
$ cd examples2/tutorial/creating_packages/other_packages/header_only
```

A header-only library is composed only of header files. That means a consumer doesn't link with any library but includes headers, so we need only one binary configuration for a header-only library.

In the *Create your first Conan package* section, we learned about the settings, and how building the recipe applying different `build_type` (Release/Debug) generates a new binary package.

As we only need one binary package, we don't need to declare the *settings* attribute. This is a basic recipe for a header-only recipe:

Listing 63: conanfile.py

```python
from conan import ConanFile
from conan.tools.files import copy


class SumConan(ConanFile):
    name = "sum"
    version = "0.1"
    # No settings/options are necessary, this is header only
    exports_sources = "include/*"
    # We can avoid copying the sources to the build folder in the cache
    no_copy_source = True

    def package(self):
        # This will also copy the "include" folder
        copy(self, "*.h", self.source_folder, self.package_folder)
```

Our header-only library is this simple function that sums two numbers:

Listing 64: include/sum.h

```cpp
inline int sum(int a, int b){
    return a + b;
}
```

The folder *examples2/tutorial/creating_packages/other_packages/header_only* in the cloned project contains a `test_package` folder with an example of an application consuming the header-only library. So we can run a `conan create .` command to build the package and test the package:

```
$ conan create .
...
[ 50%] Building CXX object CMakeFiles/example.dir/src/example.cpp.o
[100%] Linking CXX executable example
[100%] Built target example

-------- Testing the package: Running test() ----------
sum/0.1 (test package): Running test()
sum/0.1 (test package): RUN: ./example
1 + 3 = 4
```

After running the `conan create` a new binary package is created for the header-only library, and we can see how the `test_package` project can use it correctly.

We can list the binary packages created running this command:

```
$ conan list sum/0.1#:*
Local Cache:
sum
```

(continues on next page)

(continued from previous page)

```
    sum/0.1#8d9f1fb3655adcb348befcd8374c5292 (2022-12-22 17:33:45 UTC)
    PID: da39a3ee5e6b4b0d3255bfef95601890afd80709 (2022-12-22 17:33:45 UTC)
        No package info/revision was found.
```

We get one package with the package ID `da39a3ee5e6b4b0d3255bfef95601890afd80709`. Let's see what happen if we run the `conan create` but specifying `-s build_type=Debug`:

```
$ conan create . -s build_type=Debug
$ conan list sum/0.1#:*
Local Cache:
sum
    sum/0.1#8d9f1fb3655adcb348befcd8374c5292 (2022-12-22 17:34:23 UTC)
    PID: da39a3ee5e6b4b0d3255bfef95601890afd80709 (2022-12-22 17:34:23 UTC)
        No package info/revision was found.
```

Even in the `test_package` executable is built for Debug, we get the same binary package for the header-only library. This is because we didn't specify the `settings` attribute in the recipe, so the changes in the input settings (`-s build_type=Debug`) do not affect the recipe and therefore the generated binary package is always the same.

Header-only library with tests

In the previous example, we saw why a recipe header-only library shouldn't declare the `settings` attribute, but sometimes the recipe needs them to build some executable, for example, for testing the library. Nonetheless, the binary package of the header-only library should still be unique, so we are going to review how to achieve that.

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

```
$ git clone https://github.com/conan-io/examples2.git
$ cd examples2/tutorial/creating_packages/other_packages/header_only_gtest
```

We have the same header-only library that sums two numbers, but now we have this recipe:

```python
import os
from conan import ConanFile
from conan.tools.files import copy
from conan.tools.cmake import cmake_layout, CMake


class SumConan(ConanFile):
    name = "sum"
    version = "0.1"
    settings = "os", "arch", "compiler", "build_type"
    exports_sources = "include/*", "test/*"
    no_copy_source = True
    generators = "CMakeToolchain", "CMakeDeps"

    def requirements(self):
        self.test_requires("gtest/1.11.0")

    def validate(self):
        check_min_cppstd(self, 11)

    def layout(self):
        cmake_layout(self)
```

(continues on next page)

(continued from previous page)

```
    def build(self):
        if not self.conf.get("tools.build:skip_test", default=False):
            cmake = CMake(self)
            cmake.configure(build_script_folder="test")
            cmake.build()
            self.run(os.path.join(self.cpp.build.bindir, "test_sum"))

    def package(self):
        # This will also copy the "include" folder
        copy(self, "*.h", self.source_folder, self.package_folder)

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

These are the changes introduced in the recipe:

- We are introducing a `test_require` to `gtest/1.11.0`. A `test_require` is similar to a regular requirement but it is not propagated to the consumers and cannot conflict.
- `gtest` needs at least C++11 to build. So we introduced a `validate()` method calling `check_min_cppstd`.
- As we are building the `gtest` examples with CMake, we use the generators `CMakeToolchain` and `CMakeDeps`, and we declared the `cmake_layout()` to have a known/standard directory structure.
- We have a `build()` method, building the tests, but only when the standard conf `tools.build:skip_test` is not True. Use that conf as a standard way to enable/disable the testing. It is used by the helpers like `CMake` to skip the `cmake.test()` in case we implement the tests in CMake.
- We have a `package_id()` method calling `self.info.clear()`. This is internally removing the settings from the package ID calculation so we generate only one configuration for our header-only library.

We can call `conan create` to build and test our package.

```
$ conan create . -s compiler.cppstd=14 --build missing
...
Running main() from /Users/luism/.conan2/p/tmp/9bf83ef65d5ff0d6/b/googletest/
↪src/gtest_main.cc
[==========] Running 1 test from 1 test suite.
[----------] Global test environment set-up.
[----------] 1 test from SumTest
[ RUN      ] SumTest.BasicSum
[       OK ] SumTest.BasicSum (0 ms)
[----------] 1 test from SumTest (0 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test suite ran. (0 ms total)
[  PASSED  ] 1 test.
sum/0.1: Package 'da39a3ee5e6b4b0d3255bfef95601890afd80709' built
...
```

We can run `conan create` again specifying a different `compiler.cppstd` and the built package would be the same:

```
$ conan create . -s compiler.cppstd=17
...
sum/0.1: RUN: ./test_sum
Running main() from /Users/luism/.conan2/p/tmp/9bf83ef65d5ff0d6/b/googletest/
↪src/gtest_main.cc
```

(continues on next page)

(continued from previous page)

```
[==========] Running 1 test from 1 test suite.
[----------] Global test environment set-up.
[----------] 1 test from SumTest
[ RUN      ] SumTest.BasicSum
[       OK ] SumTest.BasicSum (0 ms)
[----------] 1 test from SumTest (0 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test suite ran. (0 ms total)
[  PASSED  ] 1 test.
sum/0.1: Package 'da39a3ee5e6b4b0d3255bfef95601890afd80709' built
```

Note: Once we have the `sum/0.1` binary package available (in a server, after a `conan upload`, or in the local cache), we can install it even if we don't specify input values for `os`, `arch`, ... etc. This is a new feature of Conan 2.X.

We could call `conan install --require sum/0.1` with an empty profile and would get the binary package from the server. But if we miss the binary and we need to build the package again, it will fail because of the lack of settings.

Package prebuilt binaries

There are specific scenarios in which it is necessary to create packages from existing binaries, for example from 3rd parties or binaries previously built by another process or team that is not using Conan. Under these circumstances, building from sources is not what you want.

You can package the local files in the following scenarios:

1. When you are developing your package locally and you want to quickly create a package with the built artifacts, but as you don't want to rebuild again (clean copy) your artifacts, you don't want to call **conan create**. This method will keep your local project build if you are using an IDE.
2. When you cannot build the packages from sources (when only pre-built binaries are available) and you have them in a local directory.
3. Same as 2 but you have the precompiled libraries in a remote repository.

Locally building binaries

Use the **conan new** command to create a "Hello World" C++ library example project:

```
$ conan new cmake_lib -d name=hello -d version=1.0
```

This will create a Conan package project with the following structure.

```
.
├── CMakeLists.txt
├── conanfile.py
├── include
│   └── hello.h
├── src
│   └── hello.cpp
└── test_package
```

(continues on next page)

(continued from previous page)

```
├── CMakeLists.txt
├── conanfile.py
└── src
    └── example.cpp
```

We have a `CMakeLists.txt` file in the root, an `src` folder with the `cpp` files and, an `include` folder for the headers.

They also have a `test_package/` folder to test that the exported package is working correctly.

Now, for every different configuration (different compilers, architectures, build_type. . .):

1. We call **`conan install`** to generate the `conan_toolchain.cmake` file and the `CMakeUserPresets.json` that can be used in our IDE or calling CMake (only >= 3.23).

```
$ conan install . -s build_type=Release
```

2. We build our project calling CMake, our IDE, . . . etc:

Listing 65: Linux, macOS

```
$ mkdir -p build/Release
$ cd build/Release
$ cmake ../.. -DCMAKE_BUILD_TYPE=Release -DCMAKE_TOOLCHAIN_FILE=../Release/
↪generators/conan_toolchain.cmake
$ cmake --build .
```

Listing 66: Windows

```
$ mkdir -p build
$ cd build
$ cmake ..  -DCMAKE_TOOLCHAIN_FILE=generators/conan_toolchain.cmake
$ cmake --build . --config Release
```

Note: As we are directly using our IDE or CMake to build the library, the `build()` method of the recipe is never called and could be removed.

3. We call **`conan export-pkg`** to package the built artifacts.

```
$ cd ../..
$ conan export-pkg . -s build_type=Release
...
hello/0.1: Calling package()
hello/0.1 package(): Packaged 1 '.h' file: hello.h
hello/0.1 package(): Packaged 1 '.a' file: libhello.a
...
hello/0.1: Package '54a3ab9b777a90a13e500dd311d9cd70316e9d55' created
```

Let's deep a bit more in the package method. The generated `package()` method is using `cmake.install()` to copy the artifacts from our local folders to the Conan package.

There is an alternative and generic `package()` method that could be used for any build system:

```
def package(self):
    local_include_folder = os.path.join(self.source_folder, self.cpp.source.
↪includedirs[0])
```

(continues on next page)

(continued from previous page)

```
    local_lib_folder = os.path.join(self.build_folder, self.cpp.build.libdirs[0])
    copy(self, "*.h", local_include_folder, os.path.join(self.package_folder,
↪"include"), keep_path=False)
    copy(self, "*.lib", local_lib_folder, os.path.join(self.package_folder, "lib
↪"), keep_path=False)
    copy(self, "*.a", local_lib_folder, os.path.join(self.package_folder, "lib"),␣
↪keep_path=False)
```

This `package()` method is copying artifacts from the following directories that, thanks to the layout(), will always point to the correct places:

- **os.path.join(self.source_folder, self.cpp.source.includedirs[0])** will always point to our local include folder.
- **os.path.join(self.build_folder, self.cpp.build.libdirs[0])** will always point to the location of the libraries when they are built, no matter if using a single-config CMake Generator or a multi-config one.

4. We can test the built package calling **conan test**:

```
$ conan test test_package/conanfile.py hello/0.1 -s build_type=Release

-------- Testing the package: Running test() ----------
hello/0.1 (test package): Running test()
hello/0.1 (test package): RUN: ./example
hello/0.1: Hello World Release!
  hello/0.1: __x86_64__ defined
  hello/0.1: __cplusplus199711
  hello/0.1: __GNUC__4
  hello/0.1: __GNUC_MINOR__2
  hello/0.1: __clang_major__13
  hello/0.1: __clang_minor__1
  hello/0.1: __apple_build_version__13160021
```

Now you can try to generate a binary package for `build_type=Debug` running the same steps but changing the `build_type`. You can repeat this process any number of times for different configurations.

Packaging already Pre-built Binaries

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

```
$ git clone https://github.com/conan-io/examples2.git
$ cd examples2/tutorial/creating_packages/other_packages/prebuilt_binaries
```

This is an example of scenario 2 explained in the introduction. If you have a local folder containing the binaries for different configurations you can package them using the following approach.

These are the files of our example, (be aware that the library files are only empty files so not valid libraries):

```
.
├── conanfile.py
└── vendor_hello_library
    ├── linux
    │   ├── armv8
    │   │   ├── include
    │   │   │   └── hello.h
    │   │   └── libhello.a
```

(continues on next page)

(continued from previous page)

```
│    └── x86_64
│        ├── include
│        │   └── hello.h
│        └── libhello.a
├── macos
│    ├── armv8
│    │   ├── include
│    │   │   └── hello.h
│    │   └── libhello.a
│    └── x86_64
│        ├── include
│        │   └── hello.h
│        └── libhello.a
└── windows
     ├── armv8
     │   ├── hello.lib
     │   └── include
     │       └── hello.h
     └── x86_64
         ├── hello.lib
         └── include
             └── hello.h
```

We have folders with `os` and subfolders with `arch`. This the recipe of our example:

```
import os
from conan import ConanFile
from conan.tools.files import copy


class helloRecipe(ConanFile):
    name = "hello"
    version = "0.1"
    settings = "os", "arch"

    def layout(self):
        _os = str(self.settings.os).lower()
        _arch = str(self.settings.arch).lower()
        self.folders.build = os.path.join("vendor_hello_library", _os, _arch)
        self.folders.source = self.folders.build
        self.cpp.source.includedirs = ["include"]
        self.cpp.build.libdirs = ["."]

    def package(self):
        local_include_folder = os.path.join(self.source_folder, self.cpp.source.
↪includedirs[0])
        local_lib_folder = os.path.join(self.build_folder, self.cpp.build.libdirs[0])
        copy(self, "*.h", local_include_folder, os.path.join(self.package_folder,
↪"include"), keep_path=False)
        copy(self, "*.lib", local_lib_folder, os.path.join(self.package_folder, "lib
↪"), keep_path=False)
        copy(self, "*.a", local_lib_folder, os.path.join(self.package_folder, "lib"),␣
↪keep_path=False)

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

- We are not building anything, so the `build` method is not useful here.
- We can keep the same `package` method from the previous example because the location of the artifacts is declared by the `layout()`.
- Both the source folder (with headers) and the build folder (with libraries) are in the same location, in a path that follows:

  ```
  vendor_hello_library/{os}/{arch}
  ```

- The headers are in the `include` subfolder of the `self.source_folder` (we declare it in `self.cpp.source.includedirs`).
- The libraries are in the root of the `self.build_folder` folder (we declare `self.cpp.build.libdirs = ["."]`).
- We removed the `compiler` and the `build_type` because we only have different libraries depending on the operating system and the architecture (it might be a pure C library).

Now, for each different configuration we call **conan export-pkg** command, later we can list the binaries so we can check we have one package for each precompiled library:

```
$ conan export-pkg . -s os="Linux" -s arch="x86_64"
$ conan export-pkg . -s os="Linux" -s arch="armv8"
$ conan export-pkg . -s os="Macos" -s arch="x86_64"
$ conan export-pkg . -s os="Macos" -s arch="armv8"
$ conan export-pkg . -s os="Windows" -s arch="x86_64"
$ conan export-pkg . -s os="Windows" -s arch="armv8"

$ conan list hello/0.1#:*
Local Cache:
  hello
    hello/0.1#9c7634dfe0369907f569c4e583f9bc50 (2022-12-22 17:36:39 UTC)
      PID: 522dcea5982a3f8a5b624c16477e47195da2f84f (2022-12-22 17:36:36 UTC)
        settings:
          arch=x86_64
          os=Windows
      PID: 63fead0844576fc02943e16909f08fcdddd6f44b (2022-12-22 17:36:19 UTC)
        settings:
          arch=x86_64
          os=Linux
      PID: 82339cc4d6db7990c1830d274cd12e7c91ab18a1 (2022-12-22 17:36:28 UTC)
        settings:
          arch=x86_64
          os=Macos
      PID: a0cd51c51fe9010370187244af885b0efcc5b69b (2022-12-22 17:36:39 UTC)
        settings:
          arch=armv8
          os=Windows
      PID: c93719558cf197f1df5a7f1d071093e26f0e44a0 (2022-12-22 17:36:24 UTC)
        settings:
          arch=armv8
          os=Linux
      PID: dcf68e932572755309a5f69f3cee1bede410e907 (2022-12-22 17:36:32 UTC)
        settings:
          arch=armv8
          os=Macos
```

In this example, we don't have a `test_package/` folder but you can provide one to test the packages like in the previous example.

Downloading and Packaging Pre-built Binaries

This is an example of scenario 3 explained in the introduction. If we are not building the libraries we likely have them somewhere in a remote repository. In this case, creating a complete Conan recipe, with the detailed retrieval of the binaries could be the preferred method, because it is reproducible, and the original binaries might be traced.

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

```
$ git clone https://github.com/conan-io/examples2.git
$ cd examples2/tutorial/creating_packages/other_packages/prebuilt_remote_binaries
```

Listing 67: conanfile.py

```
import os
from conan.tools.files import get, copy
from conan import ConanFile


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

    def build(self):
        base_url = "https://github.com/conan-io/libhello/releases/download/0.0.1/"

        _os = {"Windows": "win", "Linux": "linux", "Macos": "macos"}.get(str(self.
↪settings.os))
        _arch = str(self.settings.arch).lower()
        url = "{}/{}_{}.tgz".format(base_url, _os, _arch)
        get(self, url)

    def package(self):
        copy(self, "*.h", self.build_folder, os.path.join(self.package_folder,
↪"include"))
        copy(self, "*.lib", self.build_folder, os.path.join(self.package_folder,
↪"lib"))
        copy(self, "*.a", self.build_folder, os.path.join(self.package_folder, "lib
↪"))

    def package_info(self):
        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.

We only need to call **`conan create`** with different settings to generate the needed packages:

```
$ conan create . -s os="Linux" -s arch="x86_64"
$ conan create . -s os="Linux" -s arch="armv8"
$ conan create . -s os="Macos" -s arch="x86_64"
$ conan create . -s os="Macos" -s arch="armv8"
$ conan create . -s os="Windows" -s arch="x86_64"
$ conan create . -s os="Windows" -s arch="armv8"

$ conan list packages hello/0.1#:*
Local Cache:
  hello
```

(continues on next page)

(continued from previous page)

```
  hello/0.1#d8e4debf31f0b7b5ec7ff910f76f1e2a (2022-12-22 17:38:35 UTC)
    PID: 522dcea5982a3f8a5b624c16477e47195da2f84f (2022-12-22 17:38:33 UTC)
      settings:
        arch=x86_64
        os=Windows
    PID: 63fead0844576fc02943e16909f08fcdddd6f44b (2022-12-22 17:38:19 UTC)
      settings:
        arch=x86_64
        os=Linux
    PID: 82339cc4d6db7990c1830d274cd12e7c91ab18a1 (2022-12-22 17:38:27 UTC)
      settings:
        arch=x86_64
        os=Macos
    PID: a0cd51c51fe9010370187244af885b0efcc5b69b (2022-12-22 17:38:36 UTC)
      settings:
        arch=armv8
        os=Windows
    PID: c93719558cf197f1df5a7f1d071093e26f0e44a0 (2022-12-22 17:38:23 UTC)
      settings:
        arch=armv8
        os=Linux
    PID: dcf68e932572755309a5f69f3cee1bede410e907 (2022-12-22 17:38:30 UTC)
      settings:
        arch=armv8
        os=Macos
```

It is recommended to include also a small consuming project in a `test_package` folder to verify the package is correctly built, and then upload it to a Conan remote with **`conan upload`**.

The same building policies apply. Having a recipe fails if no Conan packages are created, and the **`--build`** argument is not defined. A typical approach for this kind of package could be to define a **`build_policy="missing"`**, especially if the URLs are also under the team's control. If they are external (on the internet), it could be better to create the packages and store them on your own Conan repository, so that the builds do not rely on third-party URLs being available.

Tool requires packages

In the "*Using build tools as Conan packages*" section we learned how to use a tool require to build (or help building) our project or Conan package. In this section we are going to learn how to create a recipe for a tool require.

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

```
$ git clone https://github.com/conan-io/examples2.git
$ cd examples2/tutorial/creating_packages/other_packages/tool_requires/tool
```

A simple tool require recipe

This is a recipe for a (fake) application that receiving a path returns 0 if the path is secure. We can check how the following simple recipe covers most of the `tool-require` use-cases:

Listing 68: conanfile.py

```
import os
from conan import ConanFile
```

(continues on next page)

(continued from previous page)

```
from conan.tools.cmake import CMakeToolchain, CMake, cmake_layout
from conan.tools.files import copy


class secure_scannerRecipe(ConanFile):
    name = "secure_scanner"
    version = "1.0"
    package_type = "application"

    # Binary configuration
    settings = "os", "compiler", "build_type", "arch"

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

    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):
        extension = ".exe" if self.settings_build.os == "Windows" else ""
        copy(self, "*secure_scanner{}".format(extension),
             self.build_folder, os.path.join(self.package_folder, "bin"), keep_
↪path=False)

    def package_info(self):
        self.buildenv_info.define("MY_VAR", "23")
```

There are few relevant things in this recipe:

1. It declares `package_type = "application"`, this is optional but convenient, it will indicate conan that the current package doesn't contain headers or libraries to be linked. The consumers will know that this package is an application.
2. The `package()` method is packaging the executable into the `bin/` folder, that is declared by default as a bindir: `self.cpp_info.bindirs = ["bin"]`.
3. In the `package_info()` method, we are using `self.buildenv_info` to define a environment variable `MY_VAR` that will also be available in the consumer.

Let's create a binary package for the tool_require:

```
$ conan create .
...
secure_scanner/1.0: Calling package()
secure_scanner/1.0: Copied 1 file: secure_scanner
secure_scanner/1.0 package(): Packaged 1 file: secure_scanner
...
Security Scanner: The path 'mypath' is secure!
```

Let's review the `test_package/conanfile.py`:

```
from conan import ConanFile


class secure_scannerTestConan(ConanFile):
    settings = "os", "compiler", "build_type", "arch"

    def build_requirements(self):
        self.tool_requires(self.tested_reference_str)

    def test(self):
        extension = ".exe" if self.settings_build.os == "Windows" else ""
        self.run("secure_scanner{} mypath".format(extension))
```

We are requiring the `secure_scanner` package as `tool_require` doing `self.tool_requires(self.tested_reference_str)`. In the `test()` method we are running the application, because it is available in the PATH. In the next example we are going to see why the executables from a `tool_require` are available in the consumers.

So, let's create a consumer recipe to test if we can run the `secure_scanner` application of the `tool_require` and read the environment variable. Go to the *examples2/tutorial/creating_packages/other_packages/tool_requires/consumer* folder:

Listing 69: conanfile.py

```
from conan import ConanFile

class MyConsumer(ConanFile):
    name = "my_consumer"
    version = "1.0"
    settings = "os", "arch", "compiler", "build_type"
    tool_requires = "secure_scanner/1.0"

    def build(self):
        extension = ".exe" if self.settings_build.os == "Windows" else ""
        self.run("secure_scanner{} {}".format(extension, self.build_folder))
        if self.settings_build.os != "Windows":
            self.run("echo MY_VAR=$MY_VAR")
        else:
            self.run("set MY_VAR")
```

In this simple recipe we are declaring a `tool_require` to `secure_scanner/1.0` and we are calling directly the packaged application `secure_scanner` in the `build()` method, also printing the value of the `MY_VAR` env variable.

If we build the consumer:

```
$ conan build .

-------- Installing (downloading, building) binaries... --------
secure_scanner/1.0: Already installed!

-------- Finalizing install (deploy, generators) --------
...
conanfile.py (my_consumer/1.0): RUN: secure_scanner /Users/luism/workspace/examples2/
↪tutorial/creating_packages/other_packages/tool_requires/consumer
...
```

(continues on next page)

(continued from previous page)

```
Security Scanner: The path '/Users/luism/workspace/examples2/tutorial/creating_
↪packages/other_packages/tool_requires/consumer' is secure!
...
MY_VAR=23
```

We can see that the executable returned 0 (because our folder is secure) and it printed `Security Scanner: The path is secure!` message. It also printed the "23" value assigned to `MY_VAR` but, why are these automatically available?

- The generators `VirtualBuildEnv` and `VirtualRunEnv` are automatically used.
- The `VirtualRunEnv` is reading the `tool-requires` and is creating a launcher like `conanbuildenv-release-x86_64.sh` appending all `cpp_info.bindirs` to the `PATH`, all the `cpp_info.libdirs` to the `LD_LIBRARY_PATH` environment variable and declaring each variable of `self.buildenv_info`.
- Every time conan executes the `self.run`, by default, activates the `conanbuild.sh` file before calling any command. The `conanbuild.sh` is including the `conanbuildenv-release-x86_64.sh`, so the application is in the PATH and the enviornment variable "MYVAR" has the value declared in the tool-require.

Removing settings in package_id()

With the previous recipe, if we call **conan create** with different setting like different compiler versions, we will get different binary packages with a different `package ID`. This might be convenient to, for example, keep better traceability of our tools. In this case, the <MISSING PAGE> compatibility.py plugin can help to locate the best matching binary in case Conan doesn't find the binary for our specific compiler version.

But in some cases we might want to just generate a binary taking into account only the `os`, `arch` or at most adding the `build_type` to know if the application is built for Debug or Release. We can add a `package_id()` method to remove them:

Listing 70: conanfile.py

```python
import os
from conan import ConanFile
from conan.tools.cmake import CMakeToolchain, CMake, cmake_layout
from conan.tools.files import copy


class secure_scannerRecipe(ConanFile):
    name = "secure_scanner"
    version = "1.0"
    settings = "os", "compiler", "build_type", "arch"
    ...

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

So, if we call **conan create** with different `build_type` we will get exactly the same `package_id`.

```
$ conan create .
...
Package '82339cc4d6db7990c1830d274cd12e7c91ab18a1' created
```

(continues on next page)

(continued from previous page)

```
$ conan create . -s build_type=Debug
...
Package '82339cc4d6db7990c1830d274cd12e7c91ab18a1' created
```

We got the same binary `package_id`. The second `conan create . -s build_type=Debug` created and overwrote (created a newer package revision) of the previous Release binary, because they have the same `package_id` identifier. It is typical to create only the `Release` one, and if for any reason managing both Debug and Release binaries is intended, then the approach would be not removing the `del self.info.settings.build_type`

Read more

- Toolchains (compilers)
- Usage of *self.rundenv_info*
- `settings_target`

4.3 Working with Conan repositories

We already *learned how to download and use packages* from Conan Center that is the official repository for open source Conan packages. We also *learned how to create our own packages* and store them in the Conan local cache for reusing later. In this section we cover how you can use the Conan repositories to upload your recipes and binaries and store them for later use on another machine, project, or for sharing purposes.

First we will cover how you can setup a Conan repository locally (you can skip this part if you already have a Conan remote configured). Then we will explain how to upload packages to your own repositories and how to operate when you have multiple Conan remotes configured. Finally, we will briefly cover how you can contribute to the Conan Center central repository.

4.3.1 Setting up a Conan remote

There are several options to set-up a Conan repository:

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, and can host generic artifacts (tarballs, zips, etc). Check *Artifactory Community Edition for C/C++* for more information.
- *Conan server*: Simple, free and open source, MIT licensed server that is part of the conan-io organization project. Check *Setting-up a Conan Server* for more information.

Enterprise solutions:

- **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 the Artifactory Documentation for more information. For a comparison between Artifactory editions, check the Artifactory Comparison Matrix.

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, advanced 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.

Running Artifactory CE

There are several ways to run Artifactory CE:

- **Running from a docker image**. Just run:

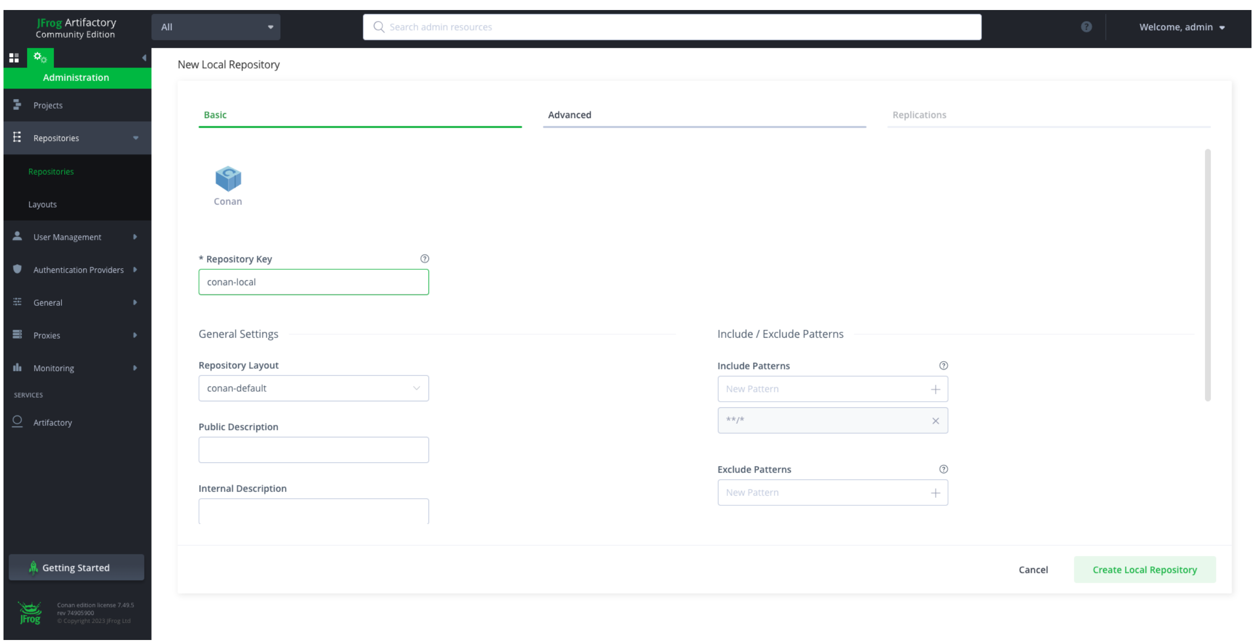
```
$ docker run --name artifactory -d -p 8081:8081 -p 8082:8082 docker.bintray.io/jfrog/artifactory-cpp-ce:latest
```

- **Download and run from zip file**. The Download Page has a link for you to follow. When the file is unzipped, launch Artifactory by double clicking the artifactory.bat(Windows) or artifactory.sh script in the *app/bin* subfolder, depending on the OS. Artifactory comes with JDK bundled, please read Artifactory requirements.

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`.

Creating and Using a Conan Repo

Navigate to Administration -> Repositories -> Repositories, then click on the "Add Repositories" button and select "Local Repository". 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" and click on "Create Local Repository". You can create multiple repositories to serve different flows, teams, or projects.



Now, let's configure the Conan client to connect with the "conan-local" repository. First add the remote to the Conan remote registry:

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

Then configure the credentials for the remote:

```
$ conan remote login artifactory <user> -p <password>
```

From now, you can upload, download, search, etc. the remote repos similarly to the other repo types.

```
$ conan upload <package_name> -r=artifactory
$ conan search "*" -r=artifactory
```

Setting-up a Conan Server

Important: This server is mainly used for testing (though it might work fine for small teams). We recommend using the free *Artifactory Community Edition for C/C++* for private development or **Artifactory Pro** as Enterprise solution.

The **Conan Server** is a free and open source server that implements Conan remote repositories. It is a very simple application, used for testing inside the Conan client and distributed as a separate pip package.

Install the **Conan Server** using pip:

```
$ pip install conan-server
```

Then you can run the server:

```
$ conan_server
 ***********************
 Using config: /Users/user/.conan_server/server.conf
 Storage: /Users/user/.conan_server/data
 Public URL: http://localhost:9300/v2
 PORT: 9300
 ***********************
 Bottle v0.12.24 server starting up (using WSGIRefServer())...
 Listening on http://0.0.0.0:9300/
 Hit Ctrl-C to quit.
```

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

See also:

- *Conan Server reference*

4.3.2 Uploading Packages

In the previous section we learned how to *set up a Conan repository*. Now we will go through the process of uploading both recipes and binaries to this remote and store them for later use on another machine, project, or for sharing purposes.

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

```
$ conan remote list
```

You can search any remote in the same way you search your Conan local cache. 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 -r=my_local_server
```

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 running on another server in your LAN:

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

Now we can check if we can 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 -c
$ conan list hello
```

Now, to install the uploaded package from the Conan repository just do:

```
$ conan install --requires=hello/1.0 -r=my_local_server
```

You can check the package existence on your local computer again with:

```
$ conan list hello
```

Read more

- *conan upload command reference*
- *conan remote command reference*
- *conan search command reference*

4.3.3 Contributing to Conan Center

Contribution of packages to ConanCenter is done via pull requests to the Github repository in https://github.com/conan-io/conan-center-index. The C3I (ConanCenter Continuous Integration) service will build binaries automatically from those pull requests, and once merged, will upload them to ConanCenter package repository.

Read more about how to submit a pull request to conan-center-index source repository.

4.4 Developing packages locally

As we learned in *previous sections* of the tutorial, the most straightforward way to work when developing a Conan package is to run a **conan create**. This means that every time it is run, Conan performs a series of costly operations in the Conan cache, such as downloading, decompressing, copying sources, and building the entire library from scratch. Sometimes, especially with large libraries, while we are developing the recipe, these operations cannot be performed every time.

This section will first show the **Conan local development flow**, that is, working on packages in your local project directory without having to export the contents of the package to the Conan cache first.

We will also cover how other packages can consume packages under development using the **editable mode**.

Finally, we will explain the **Conan package layouts** in depth, the key feature that makes it possible to work with Conan packages in the Conan cache or locally without making any changes.

4.4.1 Package Development Flow

This section introduces the **Conan local development flow**, which allows you to work on packages in your local project directory without having to export the contents of the package to the Conan cache first.

This local workflow encourages users to perform 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.

Let's use this flow for the `hello` package we created in *the previous section*.

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

```
$ git clone https://github.com/conan-io/examples2.git
$ cd examples2/tutorial/developing_packages/local_package_development_flow
```

You can check the contents of the folder:

```
.
├── conanfile.py
└── test_package
    ├── CMakeLists.txt
    ├── conanfile.py
    └── src
        └── example.cpp
```

conan source

You will generally want to start 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 relative folder is defined by the *self.folders.source* property in the *layout()* method. In this case, as we are using the pre-defined *cmake_layout* we set the value with the *src_folder* argument.

Note: In this example we are packaging a third-party library from a remote repository. In the case you have your sources beside your recipe in the same repository, running **conan source** will not be necessary for most of the cases.

Let's have a look at the recipe's *source()* and *layout()* method:

```
...

def source(self):
    # Please be aware that using the head of the branch instead of an immutable tag
    # or commit is not a good practice in general.
```

(continues on next page)

(continued from previous page)

```
    get(self, "https://github.com/conan-io/libhello/archive/refs/heads/main.zip",
        strip_root=True)

def layout(self):
    cmake_layout(self, src_folder="src")

...
```

Now run the **conan source** command and check the results:

```
$ conan source .
conanfile.py (hello/1.0): Calling source() in /Users/.../local_package_development_
↪flow/src
Downloading main.zip
conanfile.py (hello/1.0): Unzipping 3.7KB
Unzipping 100%
```

You can see that a new *src* folder has appeared containing all the *hello* library sources.

```
.
├── conanfile.py
├── src
│   ├── CMakeLists.txt
│   ├── LICENSE
│   ├── README.md
│   ├── include
│   │   └── hello.h
│   └── src
│       └── hello.cpp
└── test_package
    ├── CMakeLists.txt
    ├── conanfile.py
    └── src
        └── example.cpp
```

Now it's easy to check the sources and validate them. 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 related to downloading dependencies.

conan install

After running the **conan source** command, you can run the **conan install** command. This command will install all the recipe requirements if needed and prepare all the files necessary for building by running the `generate()` method.

We can check all the parts from our recipe that are involved in this step:

```
...

class helloRecipe(ConanFile):

    ...

    generators = "CMakeDeps"

    ...
```

(continues on next page)

(continued from previous page)

```
    def layout(self):
        cmake_layout(self, src_folder="src")

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

    ...
```

Now run the **conan install** command and check the results:

```
$ conan install .
...
-------- Finalizing install (deploy, generators) --------
conanfile.py (hello/1.0): Writing generators to ...
conanfile.py (hello/1.0): Generator 'CMakeDeps' calling 'generate()'
conanfile.py (hello/1.0): Calling generate()
...
conanfile.py (hello/1.0): Aggregating env generators
```

You can see that a new *build* folder appeared with all the files that Conan needs for building the library like a toolchain for *CMake* and several environment configuration files.

```
.
├── build
│   └── Release
│       └── generators
│           ├── CMakePresets.json
│           ├── cmakedeps_macros.cmake
│           ├── conan_toolchain.cmake
│           ├── conanbuild.sh
│           ├── conanbuildenv-release-x86_64.sh
│           ├── conanrun.sh
│           ├── conanrunenv-release-x86_64.sh
│           ├── deactivate_conanbuild.sh
│           └── deactivate_conanrun.sh
├── conanfile.py
├── src
│   ├── CMakeLists.txt
│   ├── CMakeUserPresets.json
│   ├── LICENSE
│   ├── README.md
│   ├── include
│   │   └── hello.h
│   └── src
│       └── hello.cpp
└── test_package
    ├── CMakeLists.txt
    ├── conanfile.py
    └── src
        └── example.cpp
```

Now that all the files necessary for building are generated, you can move on to testing the *build()* method.

conan build

Running the After **conan build** command will invoke the *build()* method:

```
...

class helloRecipe(ConanFile):

    ...

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

    ...
```

Let's run **conan build**:

```
$ conan build .
...
-- Conan toolchain: C++ Standard 11 with extensions ON
-- Conan toolchain: Setting BUILD_SHARED_LIBS = OFF
-- Configuring done
-- Generating done
-- Build files have been ...
conanfile.py (hello/1.0): CMake command: cmake --build ...
conanfile.py (hello/1.0): RUN: cmake --build ...
[100%] Built target hello
```

For most of the recipes, the *build()* method should be very simple, and you can also invoke the build system directly, without invoking Conan, as you have all the necessary files available for building. If you check the contents of the *src* folder, you'll find a *CMakeUserPresets.json* file that you can use to configure and build the *conan-release* preset. Let's try it:

```
$ cd src
$ cmake --preset conan-conan-release
...
-- Configuring done
-- Generating done

$ cmake --build --preset conan-conan-release
...
[100%] Built target hello
```

You can check that the results of invoking CMake directly are equivalent to the ones we got using the **conan build** command.

Note: We use CMake presets in this example. This requires CMake >= 3.23 because the "include" from `CMakeUserPresets.json` to `CMakePresets.json` is only supported since that version. If you prefer not to use presets you can use something like:

```
cmake <path> -G <CMake generator> -DCMAKE_TOOLCHAIN_FILE=<path to
conan_toolchain.cmake> -DCMAKE_BUILD_TYPE=Release
```

Conan will show the exact CMake command everytime you run `conan install` in case you can't use the presets feature.

conan export-pkg

Now that we built the package binaries locally we can also package those artifacts in the Conan local cache using the **conan export-pkg** command. Please note that this command will create the package in the Conan cache and test it running the *test_package* after that.

```
$ conan export-pkg .
conanfile.py (hello/1.0) package(): Packaged 1 '.h' file: hello.h
conanfile.py (hello/1.0) package(): Packaged 1 '.a' file: libhello.a
conanfile.py (hello/1.0): Package 'b1d267f77ddd5d10d06d2ecf5a6bc433fbb7eeed' created
conanfile.py (hello/1.0): Created package revision f09ef573c22f3919ba26ee91ae444eaa
...
conanfile.py (hello/1.0): Package folder /Users/...
conanfile.py (hello/1.0): Exported package binary
...
[ 50%] Building CXX object CMakeFiles/example.dir/src/example.cpp.o
[100%] Linking CXX executable example
[100%] Built target example

-------- Testing the package: Running test() --------
hello/1.0 (test package): Running test()
hello/1.0 (test package): RUN: ./example
hello/1.0: Hello World Release!
hello/1.0: __x86_64__ defined
hello/1.0: __cplusplus201103
hello/1.0: __GNUC__4
hello/1.0: __GNUC_MINOR__2
hello/1.0: __clang_major__14
hello/1.0: __apple_build_version__14000029
```

Now you can list the packages in the local cache and check that the `hello/1.0` package was created.

```
$ conan list hello/1.0
Local Cache
hello
    hello/1.0
```

See also:

- Reference for conan *source*, *install*, *build*, *export-pkg* and *test* commands.
- Packaging prebuilt binaries *example*

4.4.2 Packages in editable mode

The normal way of working with Conan packages is to run a `conan create` or `conan export-pkg` to store them in the local cache, so that consumers use the packages stored in the cache. In some cases, when you want to consume these packages while developing them, it can be tedious to run `conan create` each time you make changes to the package. For those cases, you can put your package in editable mode, and consumers will be able to find the headers and artifacts in your local working directory, eliminating the need for packaging.

Let's see how we can put a package in editable mode and consume it from the local working directory.

Please, first of all, 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 examples2/tutorial/developing_packages/editable_packages
```

There are 2 folders inside this project:

```
.
├── hello
│   ├── CMakeLists.txt
│   ├── conanfile.py
│   └── src
│       └── hello.cpp
└── say
    ├── CMakeLists.txt
    ├── conanfile.py
    ├── include
    │   └── say.h
    └── src
        └── say.cpp
```

- A “say” folder containing a fully fledge package, with its `conanfile.py` and its source code.
- A “hello” folder containing a simple consumer project with a `conanfile.py` and its source code, which depends on the `say/1.0` requirement.

We will put `say/1.0` in editable mode and show how the `hello` consumer can find `say/1.0` headers and binaries in its local working directory.

Put say/1.0 package in editable mode

To avoid creating the package `say/1.0` in the cache for every change, we are going to put that package in editable mode, creating **a link from the reference in the cache to the local working directory**:

```
$ conan editable add say --name=say --version=1.0
$ conan editable list
say/1.0
    Path: /Users/.../examples2/tutorial/developing_packages/editable_packages/say/
↪conanfile.py
```

From now on, every usage of `say/1.0` by any other Conan package or project will be redirected to the `/Users/../examples2/tutorial/developing_packages/editable_packages/say/conanfile.py` user folder instead of using the package from the Conan cache.

Note that the key of editable packages is a correct definition of the `layout()` of the package. Read the *package layout() section* to learn more about this method.

In this example, the `say` `conanfile.py` recipe is using the predefined `cmake_layout()` which defines the typical CMake project layout that can be different depending on the platform and generator used.

Now that the `say/1.0` package is in editable mode, let’s build it locally:

Note: We use CMake presets in this example. This requires CMake >= 3.23 because the “include” from `CMakeUserPresets.json` to `CMakePresets.json` is only supported since that version. If you prefer not to use presets you can use something like:

```
cmake <path> -G <CMake generator> -DCMAKE_TOOLCHAIN_FILE=<path to
conan_toolchain.cmake> -DCMAKE_BUILD_TYPE=Release
```

Conan will show the exact CMake command everytime you run `conan install` in case you can't use the presets feature.

```
$ cd say

# Windows: we will build 2 configurations to show multi-config
$ conan install . -s build_type=Release
$ conan install . -s build_type=Debug
$ cmake --preset conan-default
$ cmake --build --preset conan-release
$ cmake --build --preset conan-debug

# Linux, MacOS: we will only build 1 configuration
$ conan install .
$ cmake --preset conan-release
$ cmake --build --preset conan-release
```

Using say/1.0 package in editable mode

Consuming a package in editable mode is transparent from the consumer perspective. In this case we can build the `hello` application as usual:

```
$ cd ../hello

# Windows: we will build 2 configurations to show multi-config
$ conan install . -s build_type=Release
$ conan install . -s build_type=Debug
$ cmake --preset conan-default
$ cmake --build --preset conan-release
$ cmake --build --preset conan-debug
$ build\Release\hello.exe
say/1.0: Hello World Release!
...
$ build\Debug\hello.exe
say/1.0: Hello World Debug!
...

# Linux, MacOS: we will only build 1 configuration
$ conan install .
$ cmake --preset conan-release
$ cmake --build --preset conan-release
$ ./build/Release/hello
say/1.0: Hello World Release!
```

As you can see, `hello` can successfully find `say/1.0` header and library files.

Working with editable packages

Once the above steps have been completed, you can work with your build system or IDE without involving Conan and make changes to the editable packages. The consumers will use those changes directly. Let's see how this works by making a change in the `say` source code:

```
$ cd ../say
# Edit src/say.cpp and change the error message from "Hello" to "Bye"

# Windows: we will build 2 configurations to show multi-config
$ cmake --build --preset conan-release
$ cmake --build --preset conan-debug

# Linux, MacOS: we will only build 1 configuration
$ cmake --build --preset conan-release
```

And build and run the "hello" project:

```
$ cd ../hello

# Windows
$ cd build
$ cmake --build --preset conan-release
$ cmake --build --preset conan-debug
$ Release\hello.exe
say/1.0: Bye World Release!
$ Debug\hello.exe
say/1.0: Bye World Debug!

# Linux, MacOS
$ cmake --build --preset conan-release
$ ./hello
say/1.0: Bye World Release!
```

In this manner, you can develop both the `say` library and the `hello` application simultaneously without executing any Conan command in between. If you have both open in your IDE, you can simply build one after the other.

Revert the editable mode

In order to revert the editable mode just remove the link using:

```
$ conan editable remove --refs=say/1.0
```

It will remove the link (the local directory won't be affected) and all the packages consuming this requirement will get it from the cache again.

> **Warning:** Packages that are built while consuming an editable package in their upstreams can generate binaries and packages that are incompatible with the released version of the editable package. Avoid uploading these packages without re-creating them with the in-cache version of all the libraries.

4.4.3 Understanding the Conan Package layout

In the previous section, we introduced the concept of *editable packages* and mentioned that the reason they work *out of the box* when put in editable mode is due to the current definition of the information in the `layout()` method. Let's examine this feature in more detail.

In this tutorial, we will continue working with the `say/1.0` package and the `hello/1.0` consumer used in the *editable packages* tutorial.

Please, first of all, 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 examples2/tutorial/developing_packages/package_layout
```

Note: We use CMake presets in this example. This requires CMake >= 3.23 because the "include" from `CMakeUserPresets.json` to `CMakePresets.json` is only supported since that version. If you prefer not to use presets you can use something like:

```
cmake <path> -G <CMake generator> -DCMAKE_TOOLCHAIN_FILE=<path to
conan_toolchain.cmake> -DCMAKE_BUILD_TYPE=Release
```

Conan will show the exact CMake command everytime you run `conan install` in case you can't use the presets feature.

As you can see, the main folder structure is the same:

```
.
├── hello
│   ├── CMakeLists.txt
│   ├── conanfile.py
│   └── src
│       └── hello.cpp
└── say
    ├── CMakeLists.txt
    ├── conanfile.py
    ├── include
    │   └── say.h
    └── src
        └── say.cpp
```

The main difference here is that we are not using the predefined *cmake_layout()* in the `say/1.0` ConanFile, but instead, we are declaring our own custom layout. Let's see how we describe the information in the `layout()` method so that it works both when we create the package in the Conan local cache and also when the package is in editable mode.

Listing 71: say/conanfile.py

```python
import os
from conan import ConanFile
from conan.tools.cmake import CMake


class SayConan(ConanFile):
    name = "say"
    version = "1.0"

    exports_sources = "CMakeLists.txt", "src/*", "include/*"

    ...

    def layout(self):

        ## define project folder structure
```

(continues on next page)

(continued from previous page)

```
        self.folders.source = "."
        self.folders.build = os.path.join("build", str(self.settings.build_type))
        self.folders.generators = os.path.join(self.folders.build, "generators")

        ## cpp.package information is for consumers to find the package contents in␣
↪the Conan cache

        self.cpp.package.libs = ["say"]
        self.cpp.package.includedirs = ["include"] # includedirs is already set to
↪'include' by
                                                   # default, but declared for␣
↪completion
        self.cpp.package.libdirs = ["lib"]         # libdirs is already set to 'lib'␣
↪by
                                                   # default, but declared for␣
↪completion

        ## cpp.source and cpp.build information is specifically designed for editable␣
↪packages:

        # this information is relative to the source folder that is '.'
        self.cpp.source.includedirs = ["include"] # maps to ./include

        # this information is relative to the build folder that is './build/<build_
↪type>', so it will
        self.cpp.build.libdirs = ["."]  # map to ./build/<build_type> for libdirs

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

Let's review the `layout()` method. You can see that we are setting values for `self.folders` and `self.cpp`. Let's explain what these values do.

self.folders

Defines the structure of the `say` project for the source code and the folders where the files generated by Conan and the built artifacts will be located. This structure is independent of whether the package is in editable mode or exported and built in the Conan local cache. Let's define the folder structure for the `say` package:

```
say
├── CMakeLists.txt
├── conanfile.py
├── include
│   └── say.h
├── src
│   └── say.cpp
└── build
    ├── Debug                 --> Built artifacts for Debug
    │   └── generators        --> Conan generated files for Debug config
    └── Release               --> Built artifacts for Release
        └── generators        --> Conan generated files for Release config
```

- As we have our `CMakeLists.txt` in the `.` folder, `self.folders.source` is set to `.`.

- We set `self.folders.build` to be **./build/Release** or **./build/Debug** depending on the `build_type` setting. These are the folders where we want the built binaries to be located.
- The `self.folders.generators` folder is the location we set for all the files created by the Conan generators. In this case, all the files generated by the `CMakeToolchain` generator will be stored there.

Note: Please note that the values above are for a single-configuration CMake generator. To support multi-configuration generators, such as Visual Studio, you should make some changes to this layout. For a complete layout that supports both single-config and multi-config, please check the *cmake_layout()* in the Conan documentation.

self.cpp

This attribute is used to define **where consumers will find the package contents** (headers files, libraries, etc.) depending on whether the package is in editable mode or not.

cpp.package

First, we set the information for *cpp.package*. This defines the contents of the package and its location relative to the folder where the package is stored in the local cache. Please note that defining this information is equivalent to defining *self.cpp_info* in the *package_info()* method. This is the information we defined:

- `self.cpp.package.libs`: we add the `say` library so that consumers know that they should link with it. This is equivalent to declaring `self.cpp_info.libs` in the `package_info()` method.
- `self.cpp.package.libdirs`: we add the `lib` folder so that consumers know that they should search there for the libraries. This is equivalent to declaring `self.cpp_info.libdirs` in the `package_info()` method. Note that the default value for `libdirs` in both the `cpp_info` and `cpp.package` is `["lib"]` so we could have omitted that declaration.
- `self.cpp.package.includedirs`: we add the `include` folder so that consumers know that they should search there for the library headers. This is equivalent to declaring `self.cpp_info.includedirs` in the `package_info()` method. Note that the default value for `includedirs` in both the `cpp_info` and `cpp.package` is `["include"]` so we could have omitted that declaration.

To check how this information affects consumers we are going to do first do a `conan create` on the `say` package:

```
$ cd say
$ conan create . -s build_type=Release
```

When we call `conan create`, Conan moves the recipe and sources declared in the recipe to be exported to the local Cache to a recipe folder and after that, it will create a separate package folder to build the binaries and store the actual package contents. If you check in the `[YOUR_CONAN_HOME]/p` folder, you will find two new folders similar to these:

Tip: You could get the exact locations for this folders using the **conan cache** command or checking the output of the **conan create** command.

```
<YOUR_CONAN_HOME>/p
├── sayb3ea744527a91      --> folder for sources
│   └── ...
│
└── say830097e941e10      --> folder for building and storing the package binaries
```

(continues on next page)

(continued from previous page)

```
    ├── b
    │   ├── build
    │   │   └── Release
    │   ├── include
    │   │   └── say.h
    │   └── src
    │       ├── hello.cpp
    │       └── say.cpp
    └── p
        ├── include       --> defined in cpp.package.includedirs
        │   └── say.h
        └── lib           --> defined in cpp.package.libdirs
            └── libsay.a  --> defined in self.cpp.package.libs
```

You can identify there the structure we defined in the `layout()` method. If you build the `hello` consumer project now, it will search for all the headers and libraries of `say` in that folder inside the local Cache in the locations defined by `cpp.package`:

```
$ cd ../hello
$ conan install . -s build_type=Release

# Linux, MacOS
$ cmake --preset conan-release --log-level=VERBOSE
# Windows
$ cmake --preset conan-default --log-level=VERBOSE

...
-- Conan: Target declared 'say::say'
-- Conan: Library say found <YOUR_CONAN_HOME>p/say8938ceae216fc/p/lib/libsay.a
-- Created target CONAN_LIB::say_say_RELEASE STATIC IMPORTED
-- Conan: Found: <YOUR_CONAN_HOME>p/p/say8938ceae216fc/p/lib/libsay.a
-- Configuring done
...

$ cmake --build --preset conan-release
[ 50%] Building CXX object CMakeFiles/hello.dir/src/hello.cpp.o
[100%] Linking CXX executable hello
[100%] Built target hello
```

cpp.source and cpp.build

We also defined `cpp.source` and `cpp.build` attributes in our recipe. These are only used when the package is in editable mode and point to the locations that consumers will use to find headers and binaries. We defined:

- `self.cpp.source.includedirs` set to `["include"]`. This location is relative to the `self.folders.source` that we defined to `.`. In the case of editable packages, this location will be the local folder where we have our project.
- `self.cpp.build.libdirs` set to `["."]`. This location is relative to the `self.folders.build` that we defined to **./build/<build_type>**. In the case of editable packages, this location will point to **<local_folder>/build/<build_type>**.

To check how this information affects consumers, we are going to first put the `say` package in editable mode and build it locally.

```
$ cd ../say
$ conan editable add . --name=say --version=1.0
$ conan install . -s build_type=Release
$ cmake --preset conan-release
$ cmake --build --preset conan-release
```

You can check the contents of the say project's folder now, you can see that the output folders match the ones we defined with `self.folders`:

```
.
├── CMakeLists.txt
├── CMakeUserPresets.json
├── build
│   └── Release        --> defined in cpp.build.libdirs
│       ├── ...
│       ├── generators
│       │   ├── CMakePresets.json
│       │   ├── ...
│       │   └── deactivate_conanrun.sh
│       └── libsay.a  --> no need to define
├── conanfile.py
├── include            --> defined in cpp.source.includedirs
│   └── say.h
└── src
    ├── hello.cpp
    └── say.cpp
```

Now that we have the `say` package in editable mode, if we build the `hello` consumer project, it will search for all the headers and libraries of `say` in the folders defined by `cpp.source` and `cpp.build`:

```
$ cd ../hello
$ conan install . -s build_type=Release

# Linux, MacOS
$ cmake --preset conan-release --log-level=VERBOSE
# Windows
$ cmake --preset conan-default --log-level=VERBOSE

...
-- Conan: Target declared 'say::say'
-- Conan: Library say found <local_folder>/examples2/tutorial/developing_packages/
↪package_layout/say/build/Release/libsay.a
-- Conan: Found: <local_folder>/examples2/tutorial/developing_packages/package_layout/
↪say/build/Release/libsay.a
-- Configuring done
...

$ cmake --build --preset conan-release
[ 50%] Building CXX object CMakeFiles/hello.dir/src/hello.cpp.o
[100%] Linking CXX executable hello
[100%] Built target hello

$ conan editable remove --refs=say/1.0
```

Note: Please, note that we did not define `self.cpp.build.libs = ["say"]`. This is because the information set in `self.cpp.source` and `self.cpp.build` will be merged with the information set in `self.cpp.package` so that you only have to define things that change for the editable package. For the same reason, you could also omit setting `self.cpp.source.includedirs = ["include"]` but we left it there to show the use of `cpp.source`.

See also:

- Define the layout() *when you package third-party libraries*
- Define the layout() *when you have the conanfile in a subfolder*
- Define the layout() *when you want to handle multiple subprojects*

4.5 Versioning

This section of the tutorial introduces several concepts about versioning of packages.

First, explicit version updates and how to define versions of packages is explained.

Then, it will be introduced how `requires` with version ranges can help to automate updating to the latest versions.

There are some situations when recipes or source code are changed, but the version of the package is not increased. For those situations, Conan uses automatic `revisions` to be able to provide traceability and reproducibility of those changes.

Lockfiles are a common mechanism in package managers to be able to reproduce the same dependency graph later in time, even when new versions or revisions of dependencies are uploaded. Conan also provides lockfiles to be able to guarantee this reproducibility.

Finally, when different branches of a dependency graph `requires` different versions of the same package, that is called a "version conflict". The tutorial will also introduce these errors and how to address them.

4.5.1 Versions

This section explains how different versions of a given package can be created, first starting with manually changing the version attribute in the `conanfile.py` recipe, and then introducing the `set_version()` method as a mechanism to automate the definition of the package version.

Note: This section uses very simple, empty recipes without building any code, so without `build()`, `package()`, etc., to illustrate the versioning with the simplest possible recipes, and allowing the examples to run easily and to be very fast and simple. In real life, the recipes would be full-blown recipes as seen in previous sections of the tutorial, building actual libraries and packages.

Let's start with a very simple recipe:

Listing 72: conanfile.py

```
from conan import ConanFile

class pkgRecipe(ConanFile):
    name = "pkg"
    version = "1.0"

    # The recipe would export files and package them, but not really
    # necessary for the purpose of this part of the tutorial
    # exports_sources = "include/*"
```

(continues on next page)

(continued from previous page)

```
    # def package(self):
    #     ...
```

That we can create pkg/1.0 package with:

```
$ conan create .
...
pkg/1.0 .
...

$ conan list *
Local Cache
pkg
    pkg/1.0
```

If we now did some changes to the source files of this library, this would be a new version, and we could change the conanfile.py version to version = "1.1" and create the new pkg/1.1 version:

```
# Make sure you modified conanfile.py to version=1.1
$ conan create .
...
pkg/1.1 .
...

$ conan list *
Local Cache
pkg
    pkg/1.0
    pkg/1.1
```

As we can see, now we see in our cache both pkg/1.0 and pkg/1.1. The Conan cache can store any number of different versions and configurations for the same pkg package.

Automating versions

Instead of manually changing the version in conanfile.py, it is possible to automate it with 2 different approaches.

First it is possible to provide the version directly in the command line. In the example above, we could remove the version attribute from the recipe and do:

```
# Make sure you removed the version attribute in conanfile.py
$ conan create . --version=1.2
...
pkg/1.2 .
...

$ conan list *
Local Cache
pkg
    pkg/1.0
    pkg/1.1
    pkg/1.2
```

The other possibility is to use the set_version() method to define the version dynamically, for example, if the version already exists in the source code or in a text file, or it should be deduced from the git version.

Let's assume that we have a `version.txt` file in the repo, that contains just the version string `1.3`. Then, this can be done:

Listing 73: conanfile.py

```
from conan import ConanFile
from conan.tools.files import  load


class pkgRecipe(ConanFile):
    name = "pkg"

    def set_version(self):
        self.version = load(self, "version.txt")
```

```
# No need to specify the version in CLI arg or in recipe attribute
$ conan create .
...
pkg/1.3 .
...

$ conan list *
Local Cache
pkg
    pkg/1.0
    pkg/1.1
    pkg/1.2
    pkg/1.3
```

It is also possible to combine the command line version definition, falling back to reading from file if the command line argument is not provided with the following syntax:

Listing 74: conanfile.py

```
def set_version(self):
    # if self.version is already defined from CLI --version arg, it will
    # not load version.txt
    self.version = self.version or load(self, "version.txt")
```

```
# This will create the "1.4" version even if the version.txt file contains "1.3"
$ conan create . --version=1.4
...
pkg/1.4 .
...

$ conan list *
Local Cache
pkg
    pkg/1.0
    pkg/1.1
    pkg/1.2
    pkg/1.3
    pkg/1.4
```

Likewise, it is possible to obtain the version from a Git tag:

Listing 75: conanfile.py

```python
from conan import ConanFile
from conan.tools.scm import Git

class pkgRecipe(ConanFile):
    name = "pkg"

    def set_version(self):
        git = Git(self)
        tag = git.run("describe --tags")
        self.version = tag
```

```
# assuming this is a git repo, and it was tagged to 1.5
$ git init .
$ git add .
$ git commit -m "initial commit"
$ git tag 1.5
$ conan create .
    ...
    pkg/1.5 .
    ...

    $ conan list *
    Local Cache
    pkg
        pkg/1.0
        pkg/1.1
        pkg/1.2
        pkg/1.3
        pkg/1.4
        pkg/1.5
```

Note: Best practices

- We could try to use something like the branch name or the commit as the version number. However this might have some disadvantages, for example, when this package is being required, it will need a explicit `requires = "pkg/commit"` in every other package recipe requiring this one, and it might be difficult to update consumers consistently, and to know if a newer or older dependency is being used.

Requiring the new versions

When a new package version is created, if other package recipes requiring this one contain a explicit `requires`, pinning the exact version like:

Listing 76: app/conanfile.py

```python
from conan import ConanFile

class AppRecipe(ConanFile):
    name = "app"
    version = "1.0"
    requires = "pkg/1.0"
```

Then, installing or creating the `app` recipe will keep requiring and using the `pkg/1.0` version and not the newer ones. To start using the new `pkg` versions, it is necessary to explicitly update the `requires` like:

Listing 77: app/conanfile.py

```
from conan import ConanFile

class AppRecipe(ConanFile):
    name = "app"
    version = "1.0"
    requires = "pkg/1.5"
```

This process, while it achieves very good reproducibility and traceability, can be a bit tedious if we are managing a large dependency graph and we want to move forward to use the latest dependencies versions faster and with less manual intervention. To automate this, the *version-ranges* explained in the next section can be used.

4.5.2 Version ranges

In the previous section, we ended with several versions of the `pkg` package. Let's remove them and create the following simple project:

Listing 78: pkg/conanfile.py

```
from conan import ConanFile

class pkgRecipe(ConanFile):
    name = "pkg"
```

Listing 79: app/conanfile.py

```
from conan import ConanFile

class appRecipe(ConanFile):
    name = "app"
    requires = "pkg/1.0"
```

Let's create `pkg/1.0` and install `app`, to see it requires `pkg/1.0`:

```
$ conan remove "pkg*" -c
$ conan create pkg --version=1.0
... pkg/1.0 ...
$ conan install app
...
Requirements
    pkg/1.0
```

Then, if we create a new version of `pkg/1.1`, it will not automatically be used by `app`:

```
$ conan create pkg --version=1.1
... pkg/1.0 ...
# Note how this still uses the previous 1.0 version
$ conan install app
...
Requirements
    pkg/1.0
```

So we could modify `app` conanfile to explictly use the new `pkg/1.1` version, but instead of that, let's use the following version-range expression (introduced by the `[expression]` brackets):

Listing 80: app/conanfile.py

```python
from conan import ConanFile

class appRecipe(ConanFile):
    name = "app"
    requires = "pkg/[>=1.0 <2.0]"
```

When we now install the dependencies of `app`, it will automatically use the latest version in the range, even if we create a new one, without needing to modify the `app` conanfile:

```
# this will now use the newer 1.1
$ conan install app
...
Requirements
    pkg/1.1

$ conan create pkg --version=1.2
... pkg/1.2 ...
# Now it will automatically use the newest 1.2
$ conan install app
...
Requirements
    pkg/1.2
```

This holds as long as the newer version lies within the defined range, if we create a `pkg/2.0` version, `app` will not use it:

```
$ conan create pkg --version=2.0
... pkg/2.0 ...
# Conan will use the latest in the range
$ conan install app
...
Requirements
    pkg/1.2
```

Version ranges can be defined in several places:

- In `conanfile.py` recipes `requires`, `tool_requires`, `test_requires`, `python_requires`
- In `conanfile.txt` files in `[requires]`, `[tool_requires]`, `[test_requires]` sections
- In command line arguments like `--requires=` and `--tool_requires`.
- In profiles `[tool_requires]` section

Semantic versioning

The semantic versioning specification or semver, specifies that packages should be versioned using always 3 dot-separated digits like `MAJOR.MINOR.PATCH`, with very specific meanings for each digit.

Conan extends the semver specification to any number of digits, and also allows to include letters in it. This was done because during 1.X a lot of experience and feedback from users was gathered, and it became evident than in C++ the versioning scheme is often more complex, and users were demanding more flexibility, allowing versions like `1.2.3.a.8` if necessary.

The ordering of versions when necessary (for example to decide which is the latest version in a version range) is done by comparing individually each dot-separated entity in the version, from left to right. Digits will be compared numerically, so 2 < 11, and entries containing letters will be compared alphabetically (even if they also contain some numbers).

Similarly to the semver specification, Conan can manage **prereleases** and **builds** in the form: `VERSION-prerelease+build`. Conan will also order pre-releases and builds according to the same rules, and each one of them can also contain an arbitrary number of items, like `1.2.3-pre.1.2.1+build.45.a`. Note that the semver standard does not apply any ordering to builds, but Conan does, with the same logic that is used to order the main version and the pre-releases.

Important: Note that the ordering of pre-releases can be confusing at times. A pre-release happens earlier in time than the release it is qualifying. So `1.1-alpha.1` is older than `1.1`, not newer.

Range expressions

Range expressions can have comparison operators for the lower and higher bounds, separated with a space. Also, lower bounds and upper bounds in isolation are permitted, though they are generally not recommended under normal versioning schemes, specially the lower bound only. `requires = "pkg/[>=1.0 <2.0]"` will include versions like 1.0, 1.2.3 and 1.9, but will not include 0.3, 2.0 or 2.1 versions.

The tilde ~ operator can be used to define an "approximately" equal version range. `requires = "pkg/[~1]"` will include versions 1.3 and 1.8.1, but will exclude versions like 0.8 or 2.0. Likewise `requires = "pkg/[~2.5]"` will include 2.5.0 and 2.5.3, but exclude 2.1, 2.7, 2.8.

The caret ^ operator is very similar to the tilde, but allowing variability over the last defined digit. `requires = "pkg/[^1.2]"` will include 1.2.1, 1.3 and 1.51, but will exclude 1.0, 2, 2.0.

It is also possible to apply multiple conditions with the OR operator, like `requires = "pkg/[>1 <2.0 || ^3.2]"` but this kind of complex expressions is not recommended in practice and should only be used in very extreme cases.

Finally, note that pre-releases are not resolved by default. The way to include them in the range is to explicitly define it like: `requires = "pkg/[>1- <2.0]"` or more explicitly with `requires = "pkg/[>1 <2, include_prerelease=True]"`. This will include 1.5.1-pre1, but exclude 2.0-pre1.

For more information about valid range expressions go to *Requires reference*

4.5.3 Revisions

This sections introduces how doing modifications to a given recipe or source code without explicitly creating new versions, will still internally track those changes with a mechanism called revisions.

Creating different revisions

Let's start with a basic "hello" package:

```
$ mkdir hello && cd hello
$ conan remove hello* -c # clean possible existing ones
$ conan new cmake_lib -d name=hello -d version=1.0
$ conan create .
hello/1.0: Hello World Release!
...
```

We can now list the existing recipe revisions in the cache:

```
$ conan list hello/1.0#*
Local Cache
  hello
    hello/1.0
      revisions
        2475ece651f666f42c155623228c75d2 (2023-01-31 23:08:08 UTC)
```

If we now edit the `src/hello.cpp` file, to change the output message from “Hello” to “Bye”

Listing 81: hello/src/hello.cpp

```
void hello(){

    #ifdef NDEBUG
    std::cout << "hello/1.0: Bye World Release!\n";
    ...
```

So if we create the package again, without changing the version `hello/1.0`, we will get a new output:

```
$ conan create .
hello/1.0: Bye World Release!
...
```

But even if the version is the same, internally a new revision `2b547b7f20f5541c16d0b5cbcf207502` has been created.

```
$ conan list hello/1.0#*
Local Cache
  hello
    hello/1.0
      revisions
        2475ece651f666f42c155623228c75d2 (2023-01-31 23:08:08 UTC)
        2b547b7f20f5541c16d0b5cbcf207502 (2023-01-31 23:08:25 UTC)
```

This recipe **revision** is the hash of the contents of the recipe, including the `conanfile.py`, and the exported sources (`src/main.cpp`, `CMakeLists.txt`, etc., that is, all files exported in the recipe).

We can now edit the `conanfile.py`, to define the `licence` value:

Listing 82: hello/conanfile.py

```
class helloRecipe(ConanFile):
    name = "hello"
    version = "1.0"

    # Optional metadata
    license = "MIT"
    ...
```

So if we create the package again, the output will be the same, but we will also get a new revision, as the `conanfile.py` changed:

```
$ conan create .
hello/1.0: Bye World Release!
...
$ conan list hello/1.0#*
```

(continues on next page)

(continued from previous page)

```
Local Cache
  hello
    hello/1.0
      revisions
        2475ece651f666f42c155623228c75d2 (2023-01-31 23:08:08 UTC)
        2b547b7f20f5541c16d0b5cbcf207502 (2023-01-31 23:08:25 UTC)
        1d674b4349d2b1ea06aa6419f5f99dd9 (2023-01-31 23:08:34 UTC)
```

Important: The recipe **revision** is the hash of the contents. It can be changed to be the Git commit hash with `revision_mode = "scm"`. But in any case it is critical that every revision represents an immutable source, including the recipe and the source code:

- If the sources are managed with `exports_sources`, then they will be automatically be part of the hash
- If the sources are retrieved from a external location, like a downloaded tarball or a git clone, that should guarantee uniqueness, by forcing the checkout of a unique immutable tag, or a commit. Moving targets like branch names or HEAD would be broken, as revisions are considered immutable.

Any change in source code or in recipe should always imply a new revision.

Using revisions

The recipe revisions are resolved by default to the latest revision for every given version. In the case above, we could have a `chat/1.0` package that consumes the above `hello/1.0` package:

```
$ cd ..
$ mkdir chat && cd chat
$ conan new cmake_lib -d name=chat -d version=1.0 -d requires=hello/1.0
$ conan create .
...
Requirements
chat/1.0#17b45a168519b8e0ed178d822b7ad8c8 - Cache
hello/1.0#1d674b4349d2b1ea06aa6419f5f99dd9 - Cache
...
hello/1.0: Bye World Release!
chat/1.0: Hello World Release!
```

We can see that by default, it is resolving to the latest revision `1d674b4349d2b1ea06aa6419f5f99dd9`, so we also see the `hello/1.0:  Bye World` modified message.

It is possible to explicitly depend on a given revision in the recipes, so it is possible to modify the `chat/1.0` recipe to define it requires the first created revision:

Listing 83: chat/conanfile.py

```
def requirements(self):
    self.requires("hello/1.0#2475ece651f666f42c155623228c75d2")
```

So creating `chat` will now force the first revision:

```
$ conan create .
...
Requirements
chat/1.0#12f87e1b8a881da6b19cc7f229e16c76 - Cache
```

(continues on next page)

(continued from previous page)

```
hello/1.0#2475ece651f666f42c155623228c75d2 - Cache
...
hello/1.0: Hello World Release!
chat/1.0: Hello World Release!
```

Uploading revisions

The upload command will upload only the latest revision by default:

```
# upload latest revision only, all package binaries
$ conan upload hello/1.0 -c -r=myremote
```

If for some reason we want to upload all existing revisions, it is possible with:

```
# upload all revisions, all binaries for each revision
$ conan upload hello/1.0#* -c -r=myremote
```

In the server side, the latest uploaded revision becomes the latest one, and the one that will be resolved by default. For this reason, the above command uploads the different revisions in order (from older revision to latest revision), so the relative order of revisions is respected in the server side.

Note that if another machine decides to upload a revision that was created some time ago, it will still become the latest in the server side, because it is created in the server side with that time.

Package revisions

Package binaries when created also compute the hash of their contents, forming the **package revision**. But they are very different in nature to **recipe revisions**. Recipe revisions are naturally expected, every change in source code or in the recipe would cause a new recipe revision. But package binaries shouldn't have more than one **package revision**, because binaries variability would be already encoded in a unique `package_id`. Put in other words, if the recipe revision is the same (exact same input recipe and source code) and the `package_id` is the same (exact same configuration profile, settings, etc.), then that binary should be built only once.

As C and C++ build are not deterministic, it is possible that subsequents builds of the same package, without modifying anything will be creating new package revisions:

```
# Build again 2 times the latest
$ conan create .
$ conan create .
```

In some OSs like Windows, this build will not be reproducible, and the resulting artifacts will have different checksums, resulting in new package revisions:

```
$ conan list hello/1.0:*#*
Local Cache
  hello
    hello/1.0
      revisions
        1d674b4349d2b1ea06aa6419f5f99dd9 (2023-02-01 00:03:29 UTC)
          packages
            2401fa1d188d289bb25c37cfa3317e13e377a351
              revisions
                8b8c3deef5ef47a8009d4afaebfe952e (2023-01-31 23:08:40 UTC)
                8e8d380347e6d067240c4c00132d42b1 (2023-02-01 00:03:12 UTC)
```

(continues on next page)

(continued from previous page)

```
                c347faaedc1e7e3282d3bfed31700019 (2023-02-01 00:03:35 UTC)
              info
                settings
                arch: x86_64
                build_type: Release
                ...
```

By default, the package revision will also be resolved to the latest one. However, it is not possible to pin a package revision explicitly in recipes, recipes can only require down to the recipe revision as we defined above.

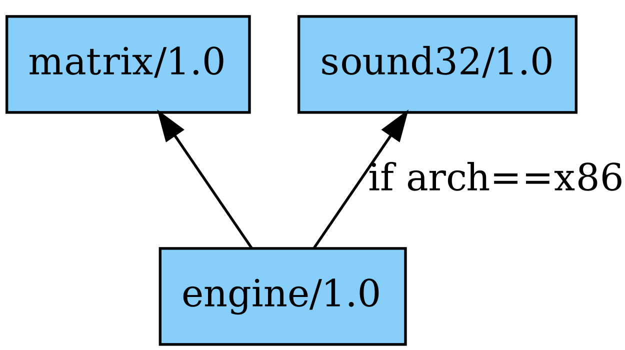
> **Warning: Best practices**
>
> Having more than 1 package revision for any given recipe revision + `package_id` is a smell or a potential bad practice. It means that something was rebuilt when it was not necessary, wasting computing and storage resources. There are ways to avoid doing it, like `conan create . --build=missing:hello*` will only build that package binary if it doesn't exist already (or running `conan graph info` can also return information of what needs to be built.)

4.5.4 Lockfiles

Lockfiles are a mechanism to achieve reproducible dependencies, even when new versions or revisions of those dependencies are created. Let's see it with a practical example, start cloning the examples2.0 repository:

```
$ git clone https://github.com/conan-io/examples2.git
$ cd examples2/tutorial/versioning/lockfiles/intro
```

In this folder we have a small project, consisting in 3 packages: a `matrix` package, emulating some mathematical library, an `engine` package emulating some game engine, and a `sound32` package, emulating a sound library for some 32bits systems. These packages are actually most empty, they do not build any code, but they are good to learn the concepts of lockfiles.



We will start by creating the first `matrix/1.0` version:

```
$ conan create matrix --version=1.0
```

Now we can check in the `engine` folder its recipe:

```
class Engine(ConanFile):
    name = "engine"
    settings = "arch"

    def requirements(self):
        self.requires("matrix/[>=1.0 <2.0]")
        if self.settings.arch == "x86":
            self.requires("sound32/[>=1.0 <2.0]")
```

Lets move to the `engine` folder and install its dependencies:

```
$ cd engine
$ conan install .
...
Requirements
    matrix/1.0#905c3f0babc520684c84127378fefdd0 - Cache
Resolved version ranges
    matrix/[>=1.0 <2.0]: matrix/1.0
```

As the `matrix/1.0` version is in the valid range, it is resolved and used. But if someone creates a new `matrix/1.1` or `1.X` version, it would also be automatically used, because it is also in the valid range. To avoid this, we will capture a "snapshot" of the current dependencies creating a `conan.lock` lockfile:

```
$ conan lock create .
$ cat conan.lock
{
    "version": "0.5",
    "requires": [
        "matrix/1.0#905c3f0babc520684c84127378fefdd0%1675278126.0552447"
    ],
    "build_requires": [],
    "python_requires": []
}
```

We can see how the created `conan.lock` lockfile contains the `matrix/1.0` version and its revision. But `sound32/1.0` is not in the lockfile, because for the default configuration profile (not `x86`), this `sound32` is not a dependency.

Now, a new `matrix/1.1` version is created:

```
$ cd ..
$ conan create matrix --version=1.1
$ cd engine
```

And see what happens when we issue a new `conan install` command for the engine:

```
$ conan install .
# equivalent to conan install . --lockfile=conan.lock
...
Requirements
   matrix/1.0#905c3f0babc520684c84127378fefdd0 - Cache
```

As we can see, the new `matrix/1.1` was not used, even if it is in the valid range! This happens because by default the `--lockfile=conan.lock` will be used if the `conan.lock` file is found. The locked `matrix/1.0` version and revision will be used to resolve the range, and the `matrix/1.1` will be ignored.

Likewise, it is possible to issue other Conan commands, and if the `conan.lock` is there, it will be used:

```
$ conan graph info . --filter=requires # --lockfile=conan.lock is implicit
# display info for matrix/1.0
$ conan create . --version=1.0 # --lockfile=conan.lock is implicit
# creates the engine/1.0 package, using matrix/1.0 as dependency
```

If using a lockfile is intended, like in CI, it is better that the argument `--lockfile=conan.lock` explicit.

Multi-configuration lockfiles

We saw above that the `engine` has a conditional dependency to the `sound32` package, in case the architecture is `x86`. That also means that such `sound32` package version was not captured in the above lockfile.

Lets create the `sound32/1.0` package first, then try to install `engine`:

```
$ cd ..
$ conan create sound32 --version=1.0
$ cd engine
$ conan install . -s arch=x86 # FAILS!
ERROR: Requirement 'sound32/[>=1.0 <2.0]' not in lockfile
```

This happens because the `conan.lock` lockfile doesn't contain a locked version for `sound32`. By default lockfiles are strict, if we are locking dependencies, a matching version inside the lockfile must be found. We can relax this assumption with the `--lockfile-partial` argument:

```
$ conan install . -s arch=x86 --lockfile-partial
...
Requirements
    matrix/1.0#905c3f0babc520684c84127378fefdd0 - Cache
    sound32/1.0#83d4b7bf607b3b60a6546f8b58b5cdd7 - Cache
Resolved version ranges
    sound32/[>=1.0 <2.0]: sound32/1.0
```

This will manage to partially lock to `matrix/1.0`, and resolve `sound32` version range as usual. But we can do better, we can extend our lockfile to also lock `sound32/1.0` version, to avoid possible disruptions caused by new `sound32` unexpected versions:

```
$ conan lock create . -s arch=x86
$ cat conan.lock
{
    "version": "0.5",
    "requires": [
        "sound32/1.0#83d4b7bf607b3b60a6546f8b58b5cdd7%1675278904.0791488",
        "matrix/1.0#905c3f0babc520684c84127378fefdd0%1675278900.0103245"
    ],
    "build_requires": [],
    "python_requires": []
}
```

Now, both `matrix/1.0` and `sound32/1.0` are locked inside our `conan.lock` lockfile. It is possible to use this lockfile for both configurations (64bits, and x86 architectures), having versions in a lockfile that are not used for a given configuration is not an issue, as long as the necessary dependencies for that configuration find a matching version in it.

Important: Lockfiles contains sorted lists of requirements, ordered by versions and revisions, so latest versions and revisions are the ones that are prioritized when resolving against a lockfile. A lockfile can contain two or more different versions of the same package, just because different version ranges require them. The sorting will provide the right logic so each range resolves to each valid versions.

If a version in the lockfile doesn't fit in a valid range, it will not be used. It is not possible for lockfiles to force a dependency that goes against what `conanfile` requires define, as they are "snapshots" of an existing/realizable dependency graph, but cannot define an "impossible" dependency graph.

Evolving lockfiles

Even if lockfiles enforce and constraint the versions that can be resolved for a graph, it doesn't mean that lockfiles cannot evolve. Actually, controlled evolution of lockfiles is paramount to important processes like Continuous Integration, when the effect of one change in the graph wants to be tested in isolation of other possible concurrent changes.

In this section we will introduce some of the basic functionality of lockfiles that allows such evolution.

First, if we would like now to introduce and test the new `matrix/1.1` version in our `engine`, without necessarily pulling many other dependencies that could have got new versions too, we could manually add `matrix/1.1` to the lockfile:

```
$ Running: conan lock add --requires=matrix/1.1
$ cat conan.lock
{
    "version": "0.5",
    "requires": [
        "sound32/1.0#83d4b7bf607b3b60a6546f8b58b5cdd7%1675278904.0791488",
        "matrix/1.1",
        "matrix/1.0#905c3f0babc520684c84127378fefdd0%1675278900.0103245"
    ],
    "build_requires": [],
    "python_requires": []
}
```

To be clear: manually adding with `conan lock add` is not necessarily a recommended flow, it is possible to automate the task with other approaches, that will be explained later. This is just an introduction to the principles and concepts.

The important idea is that now we got 2 versions of `matrix` in the lockfile, and `matrix/1.1` is before `matrix/1.0`, so for the range `matrix/[>=1.0 <2.0]`, the first one (`matrix/1.1`) would be prioritized. That means that when now the new lockfile is used, it will resolve to `matrix/1.1` version (even if a `matrix/1.2` or higher version existed in the system):

```
$ conan install . -s arch=x86 --lockfile-out=conan.lock
Requirements
    matrix/1.1#905c3f0babc520684c84127378fefdd0 - Cache
    sound32/1.0#83d4b7bf607b3b60a6546f8b58b5cdd7 - Cache
$ cat conan.lock
{
    "version": "0.5",
    "requires": [
        "sound32/1.0#83d4b7bf607b3b60a6546f8b58b5cdd7%1675278904.0791488",
```

(continues on next page)

(continued from previous page)

```
        "matrix/1.1#905c3f0babc520684c84127378fefdd0%1675278901.7527816",
        "matrix/1.0#905c3f0babc520684c84127378fefdd0%1675278900.0103245"
    ],
    "build_requires": [],
    "python_requires": []
}
```

Note that now `matrix/1.1` was resolved, and it also got its `revision` stored in the lockfile (because `--lockfile-out=conan.lock` was passed as argument).

It is true that the former `matrix/1.0` version was not used. As said above, having old versions in the lockfile that are not used is not harmful. However, if we want to prune the unused versions and revisions, we could use the `--lockfile-clean` for that purpose:

```
$ conan install . -s arch=x86 --lockfile-out=conan.lock --lockfile-clean
...
Requirements
    matrix/1.1#905c3f0babc520684c84127378fefdd0 - Cache
    sound32/1.0#83d4b7bf607b3b60a6546f8b58b5cdd7 - Cache
...
$ cat conan.lock
{
    "version": "0.5",
    "requires": [
        "sound32/1.0#83d4b7bf607b3b60a6546f8b58b5cdd7%1675278904.0791488",
        "matrix/1.1#905c3f0babc520684c84127378fefdd0%1675278901.7527816"
    ],
    "build_requires": [],
    "python_requires": []
}
```

It is relevant to note that the `-lockfile-clean` could remove locked versions in given configurations. For example, if instead of the above, the `x86_64` architecture is used, the `--lockfile-clean` will prune the "unused" `sound32`, because in that configuration is not used. It is possible to evaluate new lockfiles for every different configuration, and then merge them:

```
$ conan lock create . --lockfile-out=64.lock --lockfile-clean
$ conan lock create . -s arch=x86 --lockfile-out=32.lock --lockfile-clean
$ cat 64.lock
{
    "version": "0.5",
    "requires": [
        "matrix/1.1#905c3f0babc520684c84127378fefdd0%1675294635.6049662"
    ],
    "build_requires": [],
    "python_requires": []
}
$ cat 32.lock
{
    "version": "0.5",
    "requires": [
        "sound32/1.0#83d4b7bf607b3b60a6546f8b58b5cdd7%1675294637.9775107",
        "matrix/1.1#905c3f0babc520684c84127378fefdd0%1675294635.6049662"
    ],
    "build_requires": [],
    "python_requires": []
```

(continues on next page)

(continued from previous page)

```
}
$ conan lock merge --lockfile=32.lock --lockfile=64.lock --lockfile-out=conan.lock
$ cat conan.lock
{
    "version": "0.5",
    "requires": [
        "sound32/1.0#83d4b7bf607b3b60a6546f8b58b5cdd7%1675294637.9775107",
        "matrix/1.1#905c3f0babc520684c84127378fefdd0%1675294635.6049662"
    ],
    "build_requires": [],
    "python_requires": []
}
```

This multiple-clean + merge operation is not something that developers should do, only CI scripts, and for some advanced CI flows that will be explained later.

Read more

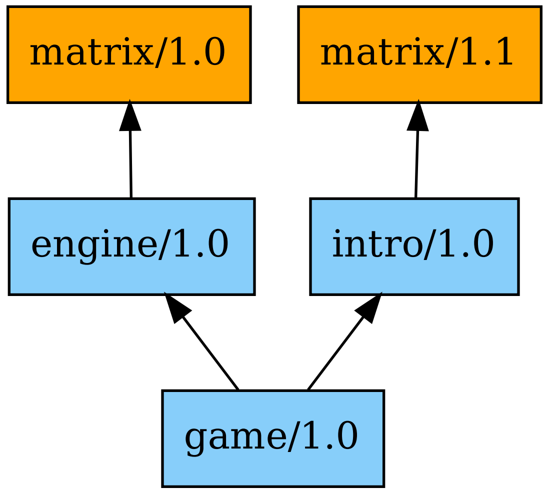
- It is possible to lock down to package revisions, but this would be not recommended for most use cases, and should only be used in extreme and problematic cases.
- Continuous Integrations links.

4.5.5 Dependencies conflicts

In a dependency graph, when different packages depends on different versions of the same package, this is called a dependency version conflict. It is relatively easy to produce one. Let's see it with a practical example, start cloning the examples2.0 repository:

```
$ git clone https://github.com/conan-io/examples2.git
$ cd examples2/tutorial/versioning/conflicts/versions
```

In this folder we have a small project, consisting in several packages: `matrix` (a math library), `engine/1.0` video game engine that depends on `matrix/1.0`, `intro/1.0`, a package implementing the intro credits and functionality for the videogame that depends on `matrix/1.1` and finally the `game` recipe that depends simultaneously on `engine/1.0` and `intro/1.0`. All these packages are actually empty, but they are enough to produce the conflicts.



Let's create the dependencies:

```
$ conan create matrix --version=1.0
$ conan create matrix --version=1.1  # note this is 1.1!
$ conan create engine --version=1.0 # depends on matrix/1.0
$ conan create intro --version=1.0 # depends on matrix/1.1
```

And when we try to install `game`, we will get the error:

```
$ conan install game
Requirements
    engine/1.0#0fe4e6890766f7b8e21f764f0049aec7 - Cache
    intro/1.0#d639998c2e55cf36d261ab319801c322 - Cache
    matrix/1.0#905c3f0babc520684c84127378fefdd0 - Cache
Graph error
    Version conflict: intro/1.0->matrix/1.1, game/1.0->matrix/1.0.
ERROR: Version conflict: intro/1.0->matrix/1.1, game/1.0->matrix/1.0.
```

This is a version conflict, and Conan will not decide automatically how to resolve the conflict, but the user should explicitly resolve such conflict.

Resolving conflicts

Of course, the most direct and straightforward way to solve such a conflict is going to the dependencies `conanfile.py` and upgrading their `requirements()` so they point now to the same version. However this might not be practical in some cases, or it might be even impossible to fix the dependencies conanfiles.

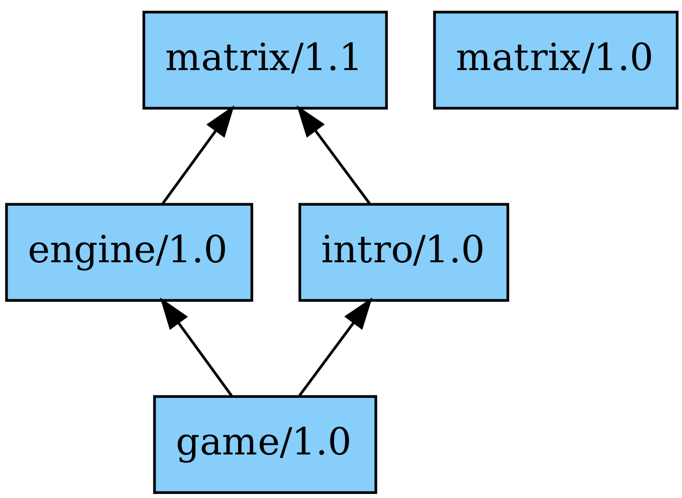
For that case, it should be the consuming `conanfile.py` the one that can resolve the conflict (in this case, `game`) by explicitly defining which version of the dependency should be used, with the following syntax:

Listing 84: game/conanfile.py

```
class Game(ConanFile):
    name = "game"
    version = "1.0"

    def requirements(self):
        self.requires("engine/1.0")
        self.requires("intro/1.0")
        self.requires("matrix/1.1", override=True)
```

This is called an `override`. The `game` package do not directly depend on `matrix`, this `requires` declaration will not introduce such a a direct dependency. But the `matrix/1.1` version will be propagated upstream in the dependency graph, overriding the `requires` of packages that do depend on any `matrix` version, forcing the consistency of the graph, as all upstream packages will now depend on `matrix/1.1`:

```
$ conan install game
...
Requirements
    engine/1.0#0fe4e6890766f7b8e21f764f0049aec7 - Cache
    intro/1.0#d639998c2e55cf36d261ab319801c322 - Cache
    matrix/1.1#905c3f0babc520684c84127378fefdd0 - Cache
```



Note: In this case, a new binary for `engine/1.0` was not necessary, but in some situations the above could fail with a `engine/1.0` "binary missing error". Because previously `engine/1.0` binaries were built against `matrix/1.`

0. If the package_id rules and configuration define that engine should be rebuilt when minor versions of the dependencies change, then it will be necessary to build a new binary for engine/1.0 that builds and links against the new matrix/1.1 dependency.

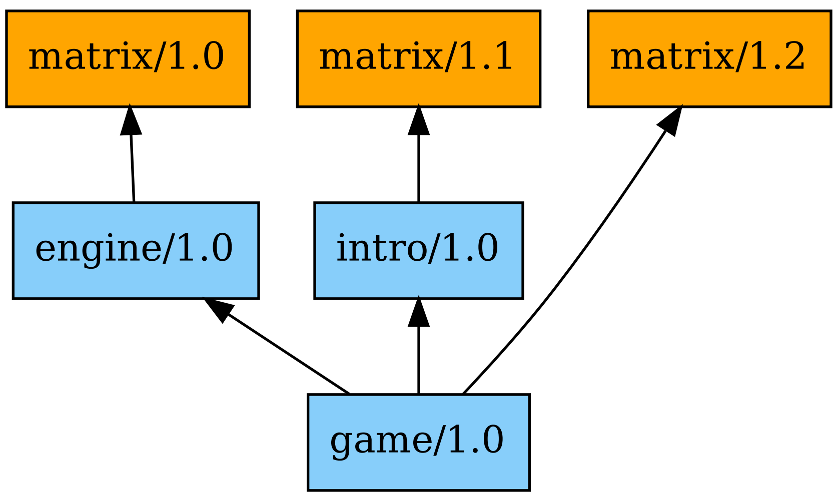
What happens if game had a direct dependency to matrix/1.2? Lets create the version:

```
$ conan create matrix --version=1.2
```

Now lets modify game/conanfile.py to introduce this as a direct dependency:

Listing 85: game/conanfile.py

```
class Game(ConanFile):
    name = "game"
    version = "1.0"

    def requirements(self):
        self.requires("engine/1.0")
        self.requires("intro/1.0")
        self.requires("matrix/1.2")
```



So intalling it will raise a conflict error again:

```
$ conan install game
...
ERROR: Version conflict: engine/1.0->matrix/1.0, game/1.0->matrix/1.2.
```

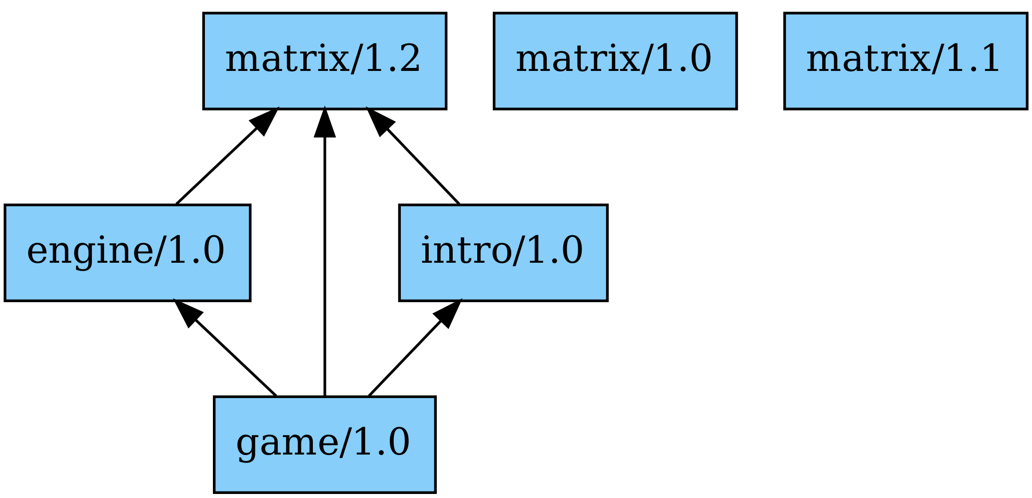
As this time, we want to respect the direct dependency between game and matrix, we will define the force=True requirement trait, to indicate that this dependency version will also be forcing the overrides upstream:

Listing 86: game/conanfile.py

```
class Game(ConanFile):
    name = "game"
    version = "1.0"

    def requirements(self):
        self.requires("engine/1.0")
        self.requires("intro/1.0")
        self.requires("matrix/1.2", force=True)
```

And that will now solve again the conflict (as commented above, note that in real applications this could mean that binaries for `engine/1.0` and `intro/1.0` would be missing, and need to be built to link against the new forced `matrix/1.2` version):

```
$ conan install game
Requirements
    engine/1.0#0fe4e6890766f7b8e21f764f0049aec7 - Cache
    intro/1.0#d639998c2e55cf36d261ab319801c322 - Cache
    matrix/1.2#905c3f0babc520684c84127378fefdd0 - Cache
```



Note: Best practices

Resolving version conflicts by overrides/forces should in general be the exception and avoided when possible, applied as a temporary workaround. The real solution is to move forward the dependencies `requires` so they naturally converge to the same versions of upstream dependencies.

Overriding options

It is possible that when there are diamond structures in a dependency graph, like the one seen above, different recipes might be defining different values for the upstream `options`. In this case, this is not directly causing a conflict, but instead the first value to be defined is the one that will be prioritized and will prevail.

In the above example, if `matrix/1.0` can be both a static and a shared library, and `engine` decides to define that it should be a static library (not really necessary, because that is already the default):

Listing 87: engine/conanfile.py

```python
class Engine(ConanFile):
    name = "engine"
    version = "1.0"
    # Not strictly necessary because this is already the matrix default
    default_options = {"matrix*:shared": False}
```

And also `intro` recipe would do the same, but instead define that it wants a shared library, and adds a `validate()` method, because for some reason the `intro` package can only be built against shared libraries and otherwise crashes:

Listing 88: intro/conanfile.py

```python
class Intro(ConanFile):
    name = "intro"
    version = "1.0"
    default_options = {"matrix*:shared": True}

    def requirements(self):
        self.requires("matrix/1.0")

    def validate(self):
        if not self.dependencies["matrix"].options.shared:
            raise ConanInvalidConfiguration("Intro package doesn't work with static␣
↪matrix library")
```

Then, this will cause an error, because as the first one to define the option value is `engine` (it is declared first in the `game` conanfile `requirements()` method). In the examples2 repository, go to the "options" folder, and create the different packages:

```
$ cd ../options
$ conan create matrix
$ conan create matrix -o matrix/*:shared=True
$ conan create engine
$ conan create intro
$ conan install game  # FAILS!
...
-------- Installing (downloading, building) binaries... --------
ERROR: There are invalid packages (packages that cannot exist for this configuration):
intro/1.0: Invalid: Intro package doesn't work with static matrix library
```

Following the same principle, the downstream consumer recipe, in this case `game` conanfile.py can define the options values, and those will be prioritized:

Listing 89: game/conanfile.py

```
class Game(ConanFile):
    name = "game"
    version = "1.0"
    default_options = {"matrix*:shared": True}

    def requirements(self):
        self.requires("engine/1.0")
        self.requires("intro/1.0")
```

And that will force now `matrix` being a shared library, no matter if `engine` defined `shared=False`, because the downstream consumers always have priority over the upstream dependencies.

```
$ conan install game
...
-------- Installing (downloading, building) binaries... --------
matrix/1.0: Already installed!
matrix/1.0: I am a shared-library library!!!
engine/1.0: Already installed!
intro/1.0: Already installed!
```

Note: Best practices

As a general rule, avoid modifying or defining values for dependencies `options` in consumers `conanfile.py`. The declared `options` defaults should be good for the majority of cases, and variations from those defaults can be defined better in profiles better.

4.6 Other important Conan features

4.6.1 python_requires

It is possible to reuse code from other recipes using the *python_requires feature*.

If you maintain many recipes for different packages that share some common logic and you don't want to repeat the code in every recipe, you can put that common code in a Conan `conanfile.py`, upload it to your server, and have other recipe conanfiles do a `python_requires = "mypythoncode/version"` to depend on it and reuse it.

CHAPTER

FIVE

INTEGRATIONS

Conan provides seamless integration with several platforms, build systems, and IDEs. Conan brings off-the-shelf support for some of the most important operating systems, including Windows, Linux, macOS, Android, and iOS. Some of the most important build systems supported by Conan include CMake, MSBuild, Meson and Autotools. In addition to build systems, Conan also provides integration with popular IDEs, such as Visual Studio and Xcode.

5.1 CMake

Conan provides different tools to integrate with CMake in a transparent way. Using these tools, the consuming `CMakeLists.txt` file does not need to be aware of Conan at all. The CMake tools also provide better IDE integration via cmake-presets.

To learn how to integrate Conan with your current CMake project you can follow the *Conan tutorial* that uses CMake along all the sections.

Please also check the reference for the CMakeDeps, CMakeToolchain, and CMake tools:

- *CMakeDeps*: responsible for generating the CMake config files for all the required dependencies of a package.
- *CMakeToolchain*: generates all the information needed for CMake to build the packages according to the information passed to Conan about things like the operating system, the compiler to use, architecture, etc. It will also generate *cmake-presets* files for easy integration with some IDEs that support this CMake feature off-the-shelf.
- *CMake* build helper is the tool used by Conan to run CMake and will pass all the arguments that CMake needs to build successfully, such as the toolchain file, build type file, and all the CMake definitions set in the recipe.

See also:

- Check the *Building your project using CMakePresets* example
- Reference for *CMakeDeps*, *CMakeToolchain* and *CMake build helper*
- *Conan tutorial*

5.2 Visual Studio

Conan provides several tools to help manage your projects using Microsoft Visual Studio. These tools can be imported from `conan.tools.microsoft` and allow for native integration with Microsoft Visual Studio, without the need to use CMake and instead directly using Visual Studio solutions, projects, and property files. The most relevant tools are:

- *MSBuildDeps*: the dependency information generator for Microsoft MSBuild build system. It will generate multiple `xxxx.props` properties files, one per dependency of a package, to be used by consumers using MSBuild or Visual Studio, just by adding the generated properties files to the solution and projects.
- *MSBuildToolchain*: the toolchain generator for MSBuild. It will generate MSBuild properties files that can be added to the Visual Studio solution projects. This generator translates the current package configuration, settings, and options, into MSBuild properties files syntax.
- *MSBuild* build helper is a wrapper around the command line invocation of MSBuild. It will abstract the calls like `msbuild "MyProject.sln" /p:Configuration=<conf> /p:Platform=<platform>` into Python method calls.

For the full list of tools under `conan.tools.microsoft` please check the *reference* section.

See also:

- Reference for *MSBuildDeps*, *MSBuildToolchain* and *MSBuild*.

5.3 Autotools

Conan provides different tools to help manage your projects using Autotools. They can be imported from `conan.tools.gnu`. The most relevant tools are:

- *AutotoolsDeps*: the dependencies generator for Autotools, which generates shell scripts containing environment variable definitions that the Autotools build system can understand.
- *AutotoolsToolchain*: the toolchain generator for Autotools, which generates shell scripts containing environment variable definitions that the Autotools build system can understand.
- *Autotools* build helper, a wrapper around the command line invocation of autotools that abstracts calls like *./configure* or *make* into Python method calls.

- *PkgConfigDeps*: the dependencies generator for *pkg-config* which generates *pkg-config* files for all the required dependencies of a package.

For the full list of tools under `conan.tools.gnu` please check the *reference* section.

See also:

- Reference for *AutotoolsDeps*, *AutotoolsToolchain*, *Autotools* and *PkgConfigDeps*.

5.4 Xcode

Conan provides different tools to integrate with Xcode IDE, providing all the necessary information about the dependencies, build options and also to build projects created with Xcode in recipes. They can be imported from `conan.tools.apple`. The most relevant tools are:

Please also check the reference for the CMakeDeps, CMakeToolchain, and CMake tools:

- *XcodeDeps*: the dependency information generator for Xcode. It will generate multiple *.xcconfig* configuration files, that can be used by consumers using xcodebuild in the command line or adding them to the Xcode IDE.
- *XcodeToolchain*: the toolchain generator for Xcode. It will generate .xcconfig configuration files that can be added to Xcode projects. This generator translates the current package configuration, settings, and options, into Xcode .xcconfig files syntax.
- *XcodeBuild* build helper is a wrapper around the command line invocation of Xcode. It will abstract the calls like `xcodebuild -project app.xcodeproj -configuration <config> -arch <arch> ...`

For the full list of tools under `conan.tools.apple` please check the *reference* section.

See also:

- Reference for *XcodeDeps*, *XcodeToolchain* and *XcodeBuild build helper*

5.5 Meson

Conan provides different tools to help manage your projects using Meson. They can be imported from `conan.tools.meson`. The most relevant tools are:

- *MesonToolchain*: generates the .ini files for Meson with the definitions of all the Meson properties related to the Conan options and settings for the current package, platform, etc. MesonToolchain normally works together with *PkgConfigDeps* to manage all the dependencies.
- *Meson* build helper, a wrapper around the command line invocation of Meson.

See also:

- Reference for *MesonToolchain* and *Meson*.
- Build a simple Meson project using Conan *example*

Build a simple Meson project using Conan

5.6 Android

Conan provides support for cross-building for Android, and it's easy to integrate with Android Studio. Please check these examples for more information on how to build your binaries for Android:

- *Cross building to Android with the NDK*
- *Integrating Conan in Android Studio*

> **Warning:** Even though there's a plugin for Visual Studio IDE and another for CLion, it's not recommended to use them right now because they're not updated for the 2.0 version yet. However, we intend to resume working on these plugins and enhance their functionality once Conan 2.0 is released.

CHAPTER

SIX

EXAMPLES

6.1 ConanFile methods examples

6.1.1 ConanFile package_info() examples

Propagating environment or configuration information to consumers

TBD

Define components for Conan packages that provide multiple libraries

TBD

6.1.2 ConanFile layout() examples

Declaring the layout when the Conanfile is inside a subfolder

Please, first of all, 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 examples2/examples/conanfile/layout/conanfile_in_subfolder
```

If we have a project intended to package the code that is in the same repo as the `conanfile.py`, but the `conanfile.py` is not in the root of the project:

```
.
├── CMakeLists.txt
├── conan
│   └── conanfile.py
├── include
│   └── say.h
└── src
    └── say.cpp
```

The `conanfile.py` would look like this:

```
import os
from conan import ConanFile
from conan.tools.files import load, copy
```

(continues on next page)

(continued from previous page)

```
from conan.tools.cmake import CMake


class PkgSay(ConanFile):
    name = "say"
    version = "1.0"
    settings = "os", "compiler", "build_type", "arch"
    generators = "CMakeToolchain"

    def layout(self):
        # The root of the project is one level above
        self.folders.root = ".."
        # The source of the project (the root CMakeLists.txt) is the source folder
        self.folders.source = "."
        self.folders.build = "build"

    def export_sources(self):
        # The path of the CMakeLists.txt and sources we want to export are one level␣
↪above
        folder = os.path.join(self.recipe_folder, "..")
        copy(self, "*.txt", folder, self.export_sources_folder)
        copy(self, "src/*.cpp", folder, self.export_sources_folder)
        copy(self, "include/*.h", folder, self.export_sources_folder)

    def source(self):
        # Check that we can see that the CMakeLists.txt is inside the source folder
        cmake_file = load(self, "CMakeLists.txt")

    def build(self):
        # Check that the build() method can also access the CMakeLists.txt in the␣
↪source folder
        path = os.path.join(self.source_folder, "CMakeLists.txt")
        cmake_file = load(self, path)

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

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

You can try and create the `say` package:

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

See also:

- Read more about the *layout method* and *how the package layout works*.

Declaring the layout when creating packages for third-party libraries

Please, first of all, 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 examples2/examples/conanfile/layout/third_party_libraries
```

If we have this project, intended to create a package for a third-party library whose code is located externally:

```
.
├── conanfile.py
└── patches
    └── mypatch
```

The `conanfile.py` would look like this:

```
...

class Pkg(ConanFile):
    name = "hello"
    version = "1.0"
    exports_sources = "patches*"

    ...

    def layout(self):
        cmake_layout(self, src_folder="src")
        # if you are declaring your own layout, just declare:
        # self.folders.source = "src"

    def source(self):
        # we are inside a "src" subfolder, as defined by layout
        # the downloaded soures will be inside the "src" subfolder
        get(self, "https://github.com/conan-io/libhello/archive/refs/heads/main.zip",
            strip_root=True)
        # Please, be aware that using the head of the branch instead of an immutable␣
↪tag
        # or commit is not a good practice in general as the branch may change the␣
↪contents

        # patching, replacing, happens here
        patch(self, patch_file=os.path.join(self.export_sources_folder, "patches/
↪mypatch"))

    def build(self):
        # If necessary, the build() method also has access to the export_sources_
↪folder
        # for example if patching happens in build() instead of source()
        #patch(self, patch_file=os.path.join(self.export_sources_folder, "patches/
↪mypatch"))
        cmake = CMake(self)
        cmake.configure()
        cmake.build()
        ...
```

We can see that the `ConanFile.export_sources_folder` attribute can provide access to the root folder of the sources:

- Locally it will be the folder where the `conanfile.py` lives
- In the cache it will be the "source" folder, that will contain a copy of `CMakeLists.txt` and `patches`, while the "source/src" folder will contain the actual downloaded sources.

We can check that everything runs fine now:

```
$ conan create .
...
Downloading main.zip
hello/1.0: Unzipping 3.7KB
Unzipping 100 %
...
[ 50%] Building CXX object CMakeFiles/hello.dir/src/hello.cpp.o
[100%] Linking CXX static library libhello.a
[100%] Built target hello
...
$ conan list hello/1.0
Local Cache
hello
    hello/1.0
```

See also:

- Read more about the *layout method* and *how the package layout works*.

Declaring the layout when we have multiple subprojects

Please, first of all, 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 examples2/examples/conanfile/layout/multiple_subprojects
```

Let's say that we have a project that contains two subprojects: *hello* and *bye*, that need to access some information that is at their same level (sibling folders). Each subproject would be a Conan package. The structure could be something similar to this:

```
.
├── bye
│   ├── CMakeLists.txt
│   ├── bye.cpp         # contains an #include "../common/myheader.h"
│   └── conanfile.py    # contains include(../common/myutils.cmake)
├── common
│   ├── myheader.h
│   └── myutils.cmake
└── hello
    ├── CMakeLists.txt # contains include(../common/myutils.cmake)
    ├── conanfile.py
    └── hello.cpp       # contains an #include "../common/myheader.h"
```

Both *hello* and *bye* subprojects needs to use some of the files located inside the `common` folder (that might be used and shared by other subprojects too), and it references them by their relative location. Note that `common` is not intended to be a Conan package. It is just some common code that will be copied into the different subproject packages.

We can use the `self.folders.root = ".."` layout specifier to locate the root of the project, then use the `self.folders.subproject = "subprojectfolder"` to relocate back most of the layout to the current subproject folder, as it would be the one containing the build scripts, sources code, etc., so other helpers like `cmake_layout()` keep working. Let's see how the *conanfile.py* of *hello* could look like:

Listing 1: ./hello/conanfile.py

```python
import os
from conan import ConanFile
from conan.tools.cmake import cmake_layout, CMake
from conan.tools.files import copy


class hello(ConanFile):
    name = "hello"
    version = "1.0"

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

    def layout(self):
        self.folders.root = ".."
        self.folders.subproject = "hello"
        cmake_layout(self)

    def export_sources(self):
        source_folder = os.path.join(self.recipe_folder, "..")
        copy(self, "hello/conanfile.py", source_folder, self.export_sources_folder)
        copy(self, "hello/CMakeLists.txt", source_folder, self.export_sources_folder)
        copy(self, "hello/hello.cpp", source_folder, self.export_sources_folder)
        copy(self, "common*", source_folder, self.export_sources_folder)

    def build(self):
        cmake = CMake(self)
        cmake.configure()
        cmake.build()
        self.run(os.path.join(self.cpp.build.bindirs[0], "hello"))
```

Let's build *hello* and check that it's building correctly, using the contents of the common folder.

```
$ conan install hello
$ conan build hello
...
[100%] Built target hello
conanfile.py (hello/1.0): RUN: ./hello
hello WORLD
```

You can also run a **`conan create`** and check that it works fine too:

```
$ conan create hello
...
[100%] Built target hello
conanfile.py (hello/1.0): RUN: ./hello
hello WORLD
```

Note: Note the importance of the `export_sources()` method, which is able to maintain the same relative layout of the `hello` and `common` folders, both in the local developer flow in the current folder, but also when those sources are copied to the Conan cache, to be built there with `conan create` or `conan install --build=hello`. This is one of the design principles of the `layout()`, the relative location of things must be consistent in the user folder and in the cache.

See also:

- Read more about the *layout method* and *how the package layout works*.

6.2 Conan extensions examples

6.2.1 Custom commands

Custom command: Clean old recipe and package revisions

Please, first of all, 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 examples2/examples/extensions/commands/clean
```

In this example we are going to see how to create/use a custom command: **conan clean**. It removes every recipe and its package revisions from the local cache or the remotes, except the latest package revision from the latest recipe one.

Note: To understand better this example, it is highly recommended to read previously the *Custom commands reference*.

Locate the command

Copy the command file `cmd_clean.py` into your `[YOUR_CONAN_HOME]/extensions/commands/` folder (create it if it's not there). If you don't know where `[YOUR_CONAN_HOME]` is located, you can run **conan config home** to check it.

Run it

Now, you should be able to see the new command in your command prompt:

```
$ conan -h
...
Custom commands
clean        Deletes (from local cache or remotes) all recipe and package revisions␣
↪but the
               latest package revision from the latest recipe revision.

$ conan clean -h
usage: conan clean [-h] [-r REMOTE] [--force]

Deletes (from local cache or remotes) all recipe and package revisions but
the latest package revision from the latest recipe revision.

optional arguments:
  -h, --help            show this help message and exit
  -r REMOTE, --remote REMOTE
                        Will remove from the specified remote
  --force               Remove without requesting a confirmation
```

Finally, if you execute **conan clean**:

```
$ conan clean
Do you want to remove all the recipes revisions and their packages ones, except the␣
↪latest package revision from the latest recipe one? (yes/no): yes
other/1.0
Removed package revision: other/1.0
↪#31da245c3399e4124e39bd4f77b5261f:da39a3ee5e6b4b0d3255bfef95601890afd80709
↪#a16985deb2e1aa73a8480faad22b722c [Local cache]
Removed recipe revision: other/1.0#721995a35b1a8d840ce634ea1ac71161 and all its␣
↪package revisions [Local cache]
hello/1.0
Removed package revision: hello/1.0
↪#9a77cdcff3a539b5b077dd811b2ae3b0:da39a3ee5e6b4b0d3255bfef95601890afd80709
↪#cee90a74944125e7e9b4f74210bfec3f [Local cache]
Removed package revision: hello/1.0
↪#9a77cdcff3a539b5b077dd811b2ae3b0:da39a3ee5e6b4b0d3255bfef95601890afd80709
↪#7cddd50952de9935d6c3b5b676a34c48 [Local cache]
libcxx/0.1
```

Nothing should happen if you run it again:

```
$ conan clean
Do you want to remove all the recipes revisions and their packages ones, except the␣
↪latest package revision from the latest recipe one? (yes/no): yes
other/1.0
hello/1.0
libcxx/0.1
```

Code tour

The `conan clean` command has the following code:

Listing 2: cmd_clean.py

```python
from conan.api.conan_api import ConanAPI
from conan.api.output import ConanOutput, Color
from conan.cli.command import OnceArgument, conan_command
from conans.client.userio import UserInput


recipe_color = Color.BRIGHT_BLUE
removed_color = Color.BRIGHT_YELLOW


@conan_command(group="Custom commands")
def clean(conan_api: ConanAPI, parser, *args):
    """
    Deletes (from local cache or remotes) all recipe and package revisions but
    the latest package revision from the latest recipe revision.
    """
    parser.add_argument('-r', '--remote', action=OnceArgument,
                        help='Will remove from the specified remote')
    parser.add_argument('--force', default=False, action='store_true',
                        help='Remove without requesting a confirmation')
    args = parser.parse_args(*args)
```

(continues on next page)

(continued from previous page)

```
    def confirmation(message):
        return args.force or ui.request_boolean(message)

    ui = UserInput(non_interactive=False)
    out = ConanOutput()
    remote = conan_api.remotes.get(args.remote) if args.remote else None
    output_remote = remote or "Local cache"

    # Getting all the recipes
    recipes = conan_api.search.recipes("*/*", remote=remote)
    if recipes and not confirmation("Do you want to remove all the recipes revisions␣
↪and their packages ones, "
                                    "except the latest package revision from the␣
↪latest recipe one?"):
        return
    for recipe in recipes:
        out.writeln(f"{str(recipe)}", fg=recipe_color)
        all_rrevs = conan_api.list.recipe_revisions(recipe, remote=remote)
        latest_rrev = all_rrevs[0] if all_rrevs else None
        for rrev in all_rrevs:
            if rrev != latest_rrev:
                conan_api.remove.recipe(rrev, remote=remote)
                out.writeln(f"Removed recipe revision: {rrev.repr_notime()} "
                            f"and all its package revisions [{output_remote}]",␣
↪fg=removed_color)
            else:
                packages = conan_api.list.packages_configurations(rrev, remote=remote)
                for package_ref in packages:
                    all_prevs = conan_api.list.package_revisions(package_ref,␣
↪remote=remote)
                    latest_prev = all_prevs[0] if all_prevs else None
                    for prev in all_prevs:
                    if prev != latest_prev:
                        conan_api.remove.package(prev, remote=remote)
                        out.writeln(f"Removed package revision: {prev.repr_notime()} [
↪{output_remote}]", fg=removed_color)
```

Let's analize the most important parts.

parser

The `parser` param is an instance of the Python command-line parsing `argparse.ArgumentParser`, so if you want to know more about its API, visit its official website.

User input and user output

Important classes to manage user input and user output:

```
ui = UserInput(non_interactive=False)
out = ConanOutput()
```

- `UserInput(non_interactive)`: class to manage user inputs. In this example we're using `ui.request_boolean("Do you want to proceed?")`, so it'll be automatically translated to `Do you want to proceed? (yes/no):` in the command prompt. **Note**: you can use `UserInput(non_interactive=conan_api.config.get("core:non_interactive"))` too.

- `ConanOutput()`: class to manage user outputs. In this example, we're using only `out.writeln(message, fg=None, bg=None)` where `fg` is the font foreground, and `bg` is the font background. Apart from that, you have some predefined methods like `out.info()`, `out.success()`, `out.error()`, etc.

Conan public API

The most important part of this example is the usage of the Conan API via `conan_api` parameter. These are some examples which are being used in this custom command:

```
conan_api.remotes.get(args.remote)
conan_api.search.recipes("*/*", remote=remote)
conan_api.list.recipe_revisions(recipe, remote=remote)
conan_api.remove.recipe(rrev, remote=remote)
conan_api.list.packages_configurations(rrev, remote=remote)
conan_api.list.package_revisions(package_ref, remote=remote)
conan_api.remove.package(prev, remote=remote)
```

- `conan_api.remotes.get(...)`: `[RemotesAPI]` Returns a RemoteRegistry given the remote name.
- `conan_api.search.recipes(...)`: `[SearchAPI]` Returns a list with all the recipes matching the given pattern.
- `conan_api.list.recipe_revisions(...)`: `[ListAPI]` Returns a list with all the recipe revisions given a recipe reference.
- `conan_api.list.packages_configurations(...)`: `[ListAPI]` Returns the list of different configurations (package_id's) for a recipe revision.
- `conan_api.list.package_revisions(...)`: `[ListAPI]` Returns the list of package revisions for a given recipe revision.
- `conan_api.remove.recipe(...)`: `[RemoveAPI]` Removes the given recipe revision.
- `conan_api.remove.package(...)`: `[RemoveAPI]` Removes the given package revision.

Besides that, it deserves especial attention these lines:

```
all_rrevs = conan_api.list.recipe_revisions(recipe, remote=remote)
latest_rrev = all_rrevs[0] if all_rrevs else None

...

packages = conan_api.list.packages_configurations(rrev, remote=remote)

...

all_prevs = conan_api.list.package_revisions(package_ref, remote=remote)
latest_prev = all_prevs[0] if all_prevs else None
```

Basically, these API calls are returning a list of recipe revisions and package ones respectively, but we're saving the first element as the latest one because these calls are getting an ordered list always.

If you want to know more about the Conan API, visit the *ConanAPI section*

6.2.2 Custom deployers

Copy sources from all your dependencies

Please, first of all, 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 examples2/examples/extensions/deployers/sources
```

In this example we are going to see how to create and use a custom deployer. This deployer copies all the source files from your dependencies and puts them into a specific output folder

Note: To better understand this example, it is highly recommended to have previously read the *Deployers* reference.

Locate the deployer

In this case, the deployer is located in the same directory as our example conanfile, but as shown in *Deployers* reference, Conan will look for the specified deployer in a few extra places in order, namely:

1. Absolute paths
2. Relative to cwd
3. In the `[CONAN_HOME]/extensions/deploy` folder
4. Built-in deployers

Run it

For our example, we have a simple recipe that lists both `zlib` and `mcap` as requirements. With the help of the `tools.build:download_source=True` conf, we can force the invocation of its `source()` method, which will ensure that sources are available even if no build needs to be carried out.

Now, you should be able to use the new deployer in both `conan install` and `conan graph` commands for any given recipe:

```
$ conan graph info . -c tools.build:download_source=True --deploy=sources_deploy
```

Inspecting the command output we can see that it copied the sources of our direct dependencies `zlib` and `mcap`, **plus** the sources of our transitive dependencies, `zstd` and `lz4` to a `dependencies_sources` folder. After this is done, extra preprocessing could be done to accomplish more specific needs.

Code tour

The **source_deploy.py** file has the following code:

Listing 3: **sources_deploy.py**

```
from conan.tools.files import copy
import os
```

(continues on next page)

(continued from previous page)

```
def deploy(graph, output_folder):
    for name, dep in graph.root.conanfile.dependencies.items():
        copy(graph.root.conanfile, "*", dep.folders.source_folder, os.path.
↪join(output_folder, "dependency_sources", str(dep)))
```

deploy()

The `deploy()` method is called by Conan, and gets both a dependency graph and an output folder path as arguments. It iterates all the dependencies of our recipe and copies every source file to their respective folders under `dependencies_sources` using *conan.tools.copy*.

6.3 Conan recipe tools examples

6.3.1 tools.cmake

CMakeToolchain: Building your project using CMakePresets

In this example we are going to see how to use `CMakeToolchain`, predefined layouts like `cmake_layout` and the `CMakePresets` CMake feature.

Let's create a basic project based on the template `cmake_exe` as an example of a C++ project:

```
$ conan new -d name=foo -d version=1.0 cmake_exe
```

Generating the toolchain

The recipe from our project declares the generator "CMakeToolchain".

We can call **conan install** to install both `Release` and `Debug` configurations. Conan will generate a `conan_toolchain.cmake` at the corresponding *generators* folder:

```
$ conan install .
$ conan install . -s build_type=Debug
```

Building the project using `CMakePresets`

A `CMakeUserPresets.json` file is generated in the same folder of your `CMakeLists.txt` file, so you can use the `--preset` argument from `cmake >= 3.23` or use an IDE that supports it.

The `CMakeUserPresets.json` is including the `CMakePresets.json` files located at the corresponding *generators* folder.

The `CMakePresets.json` contain information about the `conan_toolchain.cmake` location and even the `binaryDir` set with the output directory.

Note: We use CMake presets in this example. This requires CMake >= 3.23 because the "include" from `CMakeUserPresets.json` to `CMakePresets.json` is only supported since that version. If you prefer not to use presets you can use something like:

```
cmake <path> -G <CMake generator> -DCMAKE_TOOLCHAIN_FILE=<path to
conan_toolchain.cmake> -DCMAKE_BUILD_TYPE=Release
```

Conan will show the exact CMake command everytime you run `conan install` in case you can't use the presets feature.

If you are using a multi-configuration generator:

```
$ cmake --preset conan-default
$ cmake --build --preset conan-debug
$ build\Debug\foo.exe
foo/1.0: Hello World Release!

$ cmake --build --preset conan-release
$ build\Release\foo.exe
foo/1.0: Hello World Release!
```

If you are using a single-configuration generator:

```
$ cmake --preset conan-debug
$ cmake --build --preset conan-debug
$ ./build/Debug/foo
foo/1.0: Hello World Debug!


$ cmake --preset conan-release
$ cmake --build --preset conan-release
$ ./build/Release/foo
foo/1.0: Hello World Release!
```

Note that we didn't need to create the `build/Release` or `build/Debug` folders, as we did *in the tutorial*. The output directory is declared by the `cmake_layout()` and automatically managed by the CMake Presets feature.

This behavior is also managed automatically by Conan (with CMake >= 3.15) when you build a package in the Conan cache (with **conan create** command). The CMake >= 3.23 is not required.

Read More:

- `cmake_layout()` *reference*
- Conanfile *layout() method reference*
- Package layout tutorial *tutorial*
- Understanding *Conan package layouts*

CMakeToolchain: Extending your CMakePresets with Conan generated ones

In this example we are going to see how to extend your own CMakePresets to include Conan generated ones.

Note: We use CMake presets in this example. This requires CMake >= 3.23 because the "include" from `CMakeUserPresets.json` to `CMakePresets.json` is only supported since that version. If you prefer not to use presets you can use something like:

```
cmake <path> -G <CMake generator> -DCMAKE_TOOLCHAIN_FILE=<path to
conan_toolchain.cmake> -DCMAKE_BUILD_TYPE=Release
```

Conan will show the exact CMake command everytime you run `conan install` in case you can't use the presets feature.

Please, first of all, 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 examples2/examples/tools/cmake/cmake_toolchain/local_flow_cmake_presets
```

Please open the *conanfile.py* and check how it sets `tc.user_presets_path = 'ConanPresets.json'`. By modifying this attribute of *CMakeToolchain*, you can change the default filename of the generated preset.

```
def generate(self):
    tc = CMakeToolchain(self)
    tc.user_presets_path = 'ConanPresets.json'
    tc.generate()
    ...
```

Now you can provide your own `CMakePresets.json`, besides the `CMakeLists.txt`:

Listing 4: CMakePresets.json

```
{
"version": 4,
"include": ["./ConanPresets.json"],
"configurePresets": [
    {
        "name": "default",
        "displayName": "multi config",
        "inherits": "conan-default"
    },
    {
        "name": "release",
        "displayName": "release single config",
        "inherits": "conan-release"
    },
    {
        "name": "debug",
        "displayName": "debug single config",
        "inherits": "conan-debug"
    }
],
"buildPresets": [
    {
        "name": "multi-release",
        "configurePreset": "default",
        "configuration": "Release",
        "inherits": "conan-release"
    },
    {
        "name": "multi-debug",
        "configurePreset": "default",
        "configuration": "Debug",
        "inherits": "conan-debug"
    },
    {
        "name": "release",
```

(continues on next page)

(continued from previous page)

```
        "configurePreset": "release",
        "configuration": "Release",
        "inherits": "conan-release"
    },
    {
        "name": "debug",
        "configurePreset": "debug",
        "configuration": "Debug",
        "inherits": "conan-debug"
    }
]
}
```

Note how the `"include": ["./ConanPresets.json"],` and that every preset `inherits` a Conan generated one.

We can now install for both Release and Debug (and other configurations also, with the help of `build_folder_vars` if we want):

```
$ conan install .
$ conan install . -s build_type=Debug
```

And build and run our application, by using **our own presets** that extend the Conan generated ones:

```
# Linux (single-config, 2 configure, 2 builds)
$ cmake --preset debug
$ cmake --build --preset debug
$ ./build/Debug/foo
> Hello World Debug!

$ cmake --preset release
$ cmake --build --preset release
$ ./build/Release/foo
> Hello World Release!

# Windows VS (Multi-config, 1 configure 2 builds)
$ cmake --preset default

$ cmake --build --preset multi-debug
$ build\\Debug\\foo
> Hello World Debug!

$ cmake --build --preset multi-release
$ build\\Release\\foo
> Hello World Release!
```

6.3.2 tools.files

Patching sources

In this example we are going to see how to patch the source code. This is necessary sometimes, specially when you are creating a package for a third party library. A patch might be required in the build system scripts or even in the source code of the library if you want, for example, to apply a security patch.

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

```
$ git clone https://github.com/conan-io/examples2.git
$ cd examples/tools/files/patches
```

Patching using 'replace_in_file'

The simplest way to patch a file is using the `replace_in_file` tool in your recipe. It searches in a file the specified string and replaces it with another string.

in source() method

The source() method is called only once for all the configurations (different calls to **conan create** for different settings/options) so you should patch only in the `source()` method if the changes are common for all the configurations.

Look at the `source()` method at the `conanfile.py`:

```python
import os
from conan import ConanFile
from conan.tools.cmake import CMakeToolchain, CMake, cmake_layout
from conan.tools.files import get, replace_in_file


class helloRecipe(ConanFile):
    name = "hello"
    version = "1.0"

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

    def source(self):
        get(self, "https://github.com/conan-io/libhello/archive/refs/heads/main.zip",␣
↪strip_root=True)
        replace_in_file(self, os.path.join(self.source_folder, "src", "hello.cpp"),
↪"Hello World", "Hello Friends!")

    ...
```

We are replacing the `"Hello World"` string with "Hello Friends!". We can run `conan create .` and verify that if the replace was done:

```
$ conan create .
...
-------- Testing the package: Running test() --------
hello/1.0: Hello Friends! Release!
...
```

in build() method

In this case, we need to apply a different patch depending on the configuration (*self.settings*, *self.options. . .*), so it has to be done in the `build()` method. Let's modify the recipe to introduce a change that depends on the `self.options.shared`:

```
class helloRecipe(ConanFile):

    ...

    def source(self):
        get(self, "https://github.com/conan-io/libhello/archive/refs/heads/main.zip",␣
↪strip_root=True)

    def build(self):
        replace_in_file(self, os.path.join(self.source_folder, "src", "hello.cpp"),
                        "Hello World",
                        "Hello {} Friends!".format("Shared" if self.options.shared␣
↪else "Static"))
        cmake = CMake(self)
        cmake.configure()
        cmake.build()

    ...
```

If we call `conan create` with different `option.shared` we can check the output:

```
$ conan create .
...
hello/1.0: Hello Static Friends! Release!
...

$ conan create . -o shared=True
...
hello/1.0: Hello Shared Friends! Debug!
...
```

Patching using "patch" tool

If you have a patch file (diff between two versions of a file), you can use the `conan.tools.files.patch` tool to apply it. The rules about where to apply the patch (`source()` or `build()` methods) are the same.

We have this patch file, where we are changing again the message to say "Hello Patched World Release!":

```
--- a/src/hello.cpp
+++ b/src/hello.cpp
@@ -3,9 +3,9 @@

 void hello(){
     #ifdef NDEBUG
-    std::cout << "hello/1.0: Hello World Release!\n";
+    std::cout << "hello/1.0: Hello Patched World Release!\n";
     #else
-    std::cout << "hello/1.0: Hello World Debug!\n";
+    std::cout << "hello/1.0: Hello Patched World Debug!\n";
     #endif

     // ARCHITECTURES
```

Edit the `conanfile.py` to:

1. Import the `patch` tool.

2. Add `exports_sources` to the patch file so we have it available in the cache.
3. Call the `patch` tool.

```python
import os
from conan import ConanFile
from conan.tools.cmake import CMakeToolchain, CMake, cmake_layout
from conan.tools.files import get, replace_in_file, patch


class helloRecipe(ConanFile):
    name = "hello"
    version = "1.0"

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

    def source(self):
        get(self, "https://github.com/conan-io/libhello/archive/refs/heads/main.zip",␣
↪strip_root=True)
        patch_file = os.path.join(self.export_sources_folder, "hello_patched.patch")
        patch(self, patch_file=patch_file)

    ...
```

We can run "conan create" and see that the patch worked:

```
$ conan create .
...
-------- Testing the package: Running test() --------
hello/1.0: Hello Patched World Release!
...
```

We can also use the `conandata.yml` *introduced in the tutorial* so we can declare the patches to apply for each version:

```yaml
patches:
  "1.0":
    - patch_file: "hello_patched.patch"
```

And there are the changes we introduce in the `source()` method:

```
.. code-block:: python

    def source(self):
        get(self, "https://github.com/conan-io/libhello/archive/refs/heads/main.zip",␣
↪strip_root=True)
        patches = self.conan_data["patches"][self.version]
        for p in patches:
            patch_file = os.path.join(self.export_sources_folder, p["patch_file"])
            patch(self, patch_file=patch_file)
```

Check *patch* for more details.

If we run the **conan create**, the patch is also applied:

```
$ conan create .
...
-------- Testing the package: Running test() --------
hello/1.0: Hello Patched World Release!
...
```

Patching using "apply_conandata_patches" tool

The example above works but it is a bit complex. If you follow the same yml structure (check the *apply_conandata_patches* to see the full supported yml) you only need to call `apply_conandata_patches`:

```
from conan import ConanFile
from conan.tools.cmake import CMakeToolchain, CMake, cmake_layout
from conan.tools.files import get, apply_conandata_patches


class helloRecipe(ConanFile):
    name = "hello"
    version = "1.0"

    ...

    def source(self):
        get(self, "https://github.com/conan-io/libhello/archive/refs/heads/main.zip",␣
↪strip_root=True)
        apply_conandata_patches(self)
```

Let's check if the patch is also applied:

```
$ conan create .
...
-------- Testing the package: Running test() --------
hello/1.0: Hello Patched World Release!
...
```

6.3.3 tools.meson

Build a simple Meson project using Conan

In this example, we are going to create a string compressor application that uses one of the most popular C++ libraries: Zlib.

Note: This example is based on the main *Build a simple CMake project using Conan* tutorial. So we highly recommend reading it before trying out this one.

We'll use Meson as build system and pkg-config as helper tool in this case, so you should get them installed before going forward with this example.

Please, at 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 examples2/examples/tools/meson/mesontoolchain/simple_meson_project
```

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

```
.
├── meson.build
└── src
    └── main.c
```

This project contains a basic *meson.build* 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 5: **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());

    return EXIT_SUCCESS;
}
```

Also, the contents of *meson.build* are:

Listing 6: **meson.build**

```
project('tutorial', 'c')
zlib = dependency('zlib', version : '1.2.11')
executable('compressor', 'src/main.c', dependencies: zlib)
```

Let's create a *conanfile.txt* with the following content to install **Zlib**:

Listing 7: **conanfile.txt**

```
[requires]
zlib/1.2.11

[generators]
PkgConfigDeps
MesonToolchain
```

In this case, we will use *PkgConfigDeps* to generate information about where the **Zlib** library files are installed thanks to the **.pc* files and *MesonToolchain* to pass build information to *Meson* using a *conan_meson_[native|cross].ini* file that describes the native/cross compilation environment, which in this case is a *conan_meson_native.ini* one.

We will use Conan to install **Zlib** and generate the files that Meson needs to find this library and build our project. We will generate those files in the folder *build*. To do that, run:

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

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

Listing 8: Windows

```
$ cd build
$ meson setup --native-file conan_meson_native.ini .. meson-src
$ meson compile -C meson-src
$ meson-src\compressor.exe
Uncompressed size is: 233
Compressed size is: 147
ZLIB VERSION: 1.2.11
```

Listing 9: Linux, macOS

```
$ cd build
$ meson setup --native-file conan_meson_native.ini .. meson-src
$ meson compile -C meson-src
$ ./meson-src/compressor
Uncompressed size is: 233
Compressed size is: 147
ZLIB VERSION: 1.2.11
```

6.4 Cross-building examples

6.4.1 Cross building to Android with the NDK

In this example, we are going to see how to cross-build a Conan package to Android.

First of all, download the Android NDK from the download page and unzip it. In MacOS you can also install it with `brew install android-ndk`.

Then go to the `profiles` folder in the conan config home directory (check it running **`conan config home`**) and create a file named `android` with the following contents:

```
include(default)

```

(continues on next page)

(continued from previous page)

```
[settings]
os=Android
os.api_level=21
arch=armv8
compiler=clang
compiler.version=12
compiler.libcxx=c++_static
compiler.cppstd=14

[conf]
tools.android:ndk_path=/usr/local/share/android-ndk
```

You might need to modify:

- `compiler.version`: Check the NDK documentation or find a `bin` folder containing the compiler executables like `x86_64-linux-android31-clang`. In a Macos installation it is found in the NDK path + `toolchains/llvm/prebuilt/darwin-x86_64/bin`. Run `./x86_64-linux-android31-clang --version` to check the running `clang` version and adjust the profile.
- `compiler.libcxx`: The supported values are `c++_static` and `c++_shared`.
- `compiler.cppstd`: The C++ standard version, adjust as your needs.
- `os.api_level`: You can check here the usage of each Android Version/API level and choose the one that fits better with your requirements. This is typically a balance between new features and more compatible applications.
- `arch`: There are several architectures supported by Android: `x86`, `x86_64`, `armv7`, and `armv8`.
- `tools.android:ndk_path` conf: Write the location of the unzipped NDK.

Use the **conan new** command to create a "Hello World" C++ library example project:

```
$ conan new cmake_lib -d name=hello -d version=1.0
```

Then we can specify the `android` profile and our hello library will be built for Android:

```
$ conan create . --profile android

[ 50%] Building CXX object CMakeFiles/hello.dir/src/hello.cpp.o
[100%] Linking CXX static library libhello.a
[100%] Built target hello
...
[ 50%] Building CXX object CMakeFiles/example.dir/src/example.cpp.o
[100%] Linking CXX executable example
[100%] Built target example
```

Both the library and the `test_package` executable are built for Android, so we cannot use them in our local computer.

Unless you have access to a *root* Android device, running the test application or using the built library is not possible directly so it is more common to build an Android application that uses the `hello` library.

Read more

- Check the example *Integrating Conan in Android Studio* to know how to use your c++ libraries in a native Android application.

- Check the tutorial *How to cross-compile your applications using Conan*.

6.4.2 Integrating Conan in Android Studio

At the *Cross building to Android with the NDK* we learned how to build a package for Android using the NDK. In this example we are going to learn how to do it with the Android Studio and how to use the libraries in a real Android application.

Creating a new project

First of all, download and install the Android Studio IDE.

Then create a `new project` selecting `Native C++` from the templates.

In the next wizard window, select a name for your application, for example *MyConanApplication*, you can leave the "Minimum SDK" with the suggested value (21 in our case), but remember the value as we are using it later in the Conan profile at `os.api_level``

Select a "C++ Standard" in the next window, again, remember the choice as later we should use the same in the profile at `compiler.cppstd`.

In the project generated with the wizard we have a folder `cpp` with a `native-lib.cpp`. We are going to modify that file to use `zlib` and print a message with the used `zlib` version. Copy only the highlighted lines, it is important to keep the function name.

Listing 10: native-lib.cpp

```cpp
#include <jni.h>
#include <string>
#include "zlib.h"

extern "C" JNIEXPORT jstring JNICALL
Java_com_example_myconanapp_MainActivity_stringFromJNI(
        JNIEnv* env,
        jobject /* this */) {
    std::string hello = "Hello from C++, zlib version: ";
    hello.append(zlibVersion());
    return env->NewStringUTF(hello.c_str());
}
```

Now we are going to learn how to introduce a requirement to the `zlib` library and how to prepare our project.

Introducing dependencies with Conan

conanfile.txt

We need to provide the `zlib` package with Conan. Create a file `conanfile.txt` in the `cpp` folder:

Listing 11: conanfile.txt

```
[requires]
zlib/1.2.12

[generators]
CMakeToolchain
```

(continues on next page)

(continued from previous page)

```
CMakeDeps

[layout]
cmake_layout
```

build.gradle

We are going to automate calling `conan install` before building the Android project, so the requires are prepared, open the `build.gradle` file in the `My_Conan_App.app` (Find it in the *Gradle Scripts* section of the Android project view). Paste the `task conanInstall` contents after the `plugins` and before the `android` elements:

Listing 12: build.gradle

```
plugins {
  ...
}


task conanInstall {
    def buildDir = new File("app/build")
    buildDir.mkdirs()
    ["Debug", "Release"].each { String build_type ->
        ["armv7", "armv8", "x86", "x86_64"].each { String arch ->
            def cmd = "conan install " +
                    "../src/main/cpp --profile android -s build_type="+ build_type +"␣
↪-s arch=" + arch +
                    " --build missing -c tools.cmake.cmake_layout:build_folder_vars=[
↪'settings.arch']"
            print(">> ${cmd} \n")

            def sout = new StringBuilder(), serr = new StringBuilder()
            def proc = cmd.execute(null, buildDir)
            proc.consumeProcessOutput(sout, serr)
            proc.waitFor()
            println "$sout $serr"
            if (proc.exitValue() != 0) {
                throw new Exception("out> $sout err> $serr" + "\nCommand: ${cmd}")
            }
        }
    }
}

android {
    compileSdk 32

    defaultConfig {

...
```

The `conanInstall` task is calling **`conan install`** for Debug/Release and for each architecture we want to build, you can adjust these values to match your requirements.

If we focus on the `conan install` task we can see:

1. We are passing a `--profile android`, so we need to create the proile. Go to the `profiles` folder in the conan config home directory (check it running **`conan config home`**) and create a file named `android`

with the following contents:

```
include(default)

[settings]
os=Android
os.api_level=21
compiler=clang
compiler.version=12
compiler.libcxx=c++_static
compiler.cppstd=14

[conf]
tools.android:ndk_path=/Users/luism/Library/Android/sdk/ndk/21.4.7075529/
```

You might need to modify:

- `tools.android:ndk_path` conf: The location of the NDK provided by Android Studio. You should be able to see the path to the NDK if you open the `cpp/includes` folder in your IDE.
- `compiler.version`: Check the NDK documentation or find a `bin` folder containing the compiler executables like `x86_64-linux-android31-clang`. In a Macos installation it is found in the NDK path + `toolchains/llvm/prebuilt/darwin-x86_64/bin`. Run `./x86_64-linux-android31-clang --version` to check the running `clang` version and adjust the profile.
- `compiler.libcxx`: The supported values are `c++_static` and `c++_shared`.
- `compiler.cppstd`: The C++ standard version, this should be the value you selected in the Wizard.
- `os.api_level`: Use the same value you selected in the Wizard.

2. We are passing `-c tools.cmake.cmake_layout:build_folder_vars=['settings.arch']`, thanks to that, Conan will create a different folder for the specified `settings.arch` so we can have all the configurations available at the same time.

To make Conan work we need to pass CMake a custom toolchain. We can do it introducing a single line in the same file, in the `android/defaultConfig/externalNativeBuild/cmake` element:

Listing 13: build.gradle

```
android {
    compileSdk 32

    defaultConfig {
        applicationId "com.example.myconanapp"
        minSdk 21
        targetSdk 21
        versionCode 1
        versionName "1.0"

        testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"
        externalNativeBuild {
            cmake {
                cppFlags '-v'
                arguments("-DCMAKE_TOOLCHAIN_FILE=conan_android_toolchain.cmake")
            }
        }
```

conan_android_toolchain.cmake

Create a file called `conan_android_toolchain.cmake` in the `cpp` folder, that file will be responsible of including the right toolchain depending on the `ANDROID_ABI` variable that indicates the build configuration that the IDE is currently running:

Listing 14: conan_android_toolchain.cmake

```
if(${ANDROID_ABI} STREQUAL "x86_64")
    include("${CMAKE_CURRENT_LIST_DIR}/build/x86_64/generators/conan_toolchain.cmake")
elseif(${ANDROID_ABI} STREQUAL "x86")
    include("${CMAKE_CURRENT_LIST_DIR}/build/x86/generators/conan_toolchain.cmake")
elseif(${ANDROID_ABI} STREQUAL "arm64-v8a")
    include("${CMAKE_CURRENT_LIST_DIR}/build/armv8/generators/conan_toolchain.cmake")
elseif(${ANDROID_ABI} STREQUAL "armeabi-v7a")
    include("${CMAKE_CURRENT_LIST_DIR}/build/armv7/generators/conan_toolchain.cmake")
else()
    message(FATAL "Not supported configuration")
endif()
```

CMakeLists.txt

Finally, we need to modify the `CMakeLists.txt` to link with the `zlib` library:

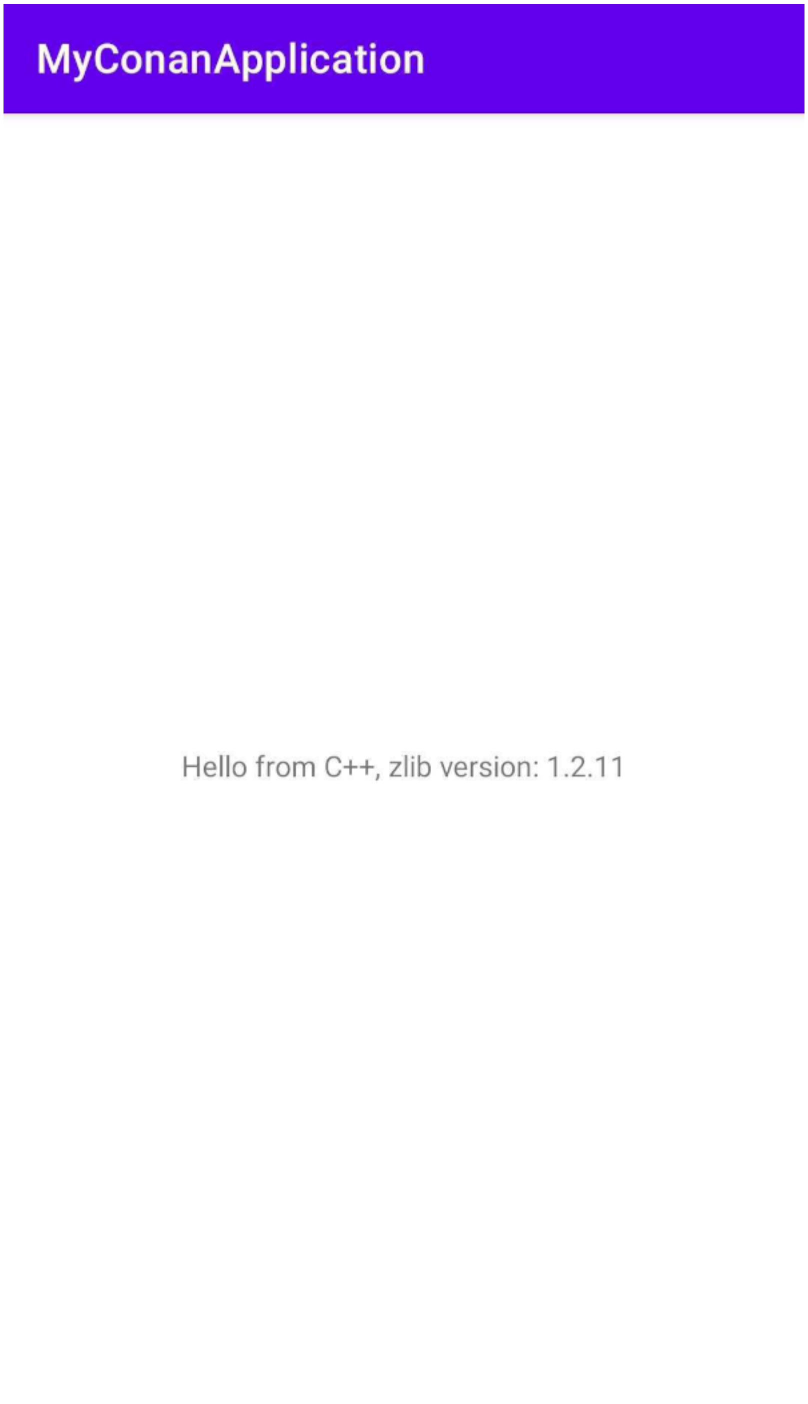
Listing 15: CMakeLists.txt

```
cmake_minimum_required(VERSION 3.18.1)
project("myconanapp")
add_library(myconanapp SHARED native-lib.cpp)

find_library(log-lib log)

find_package(ZLIB CONFIG)

target_link_libraries(myconanapp ${log-lib} ZLIB::ZLIB)
```

Building the application

If we build our project we can see that *conan install* is called multiple times building the different configurations of `zlib`.

Then if we run the application in a Virtual Device or in a real device pairing it with the QR code we can see:



Once we have our project configured, it is very easy to change our dependencies and keep developing the application, for example, we can edit the `conanfile.txt` file and change the `zlib` to the version `1.12.2`:

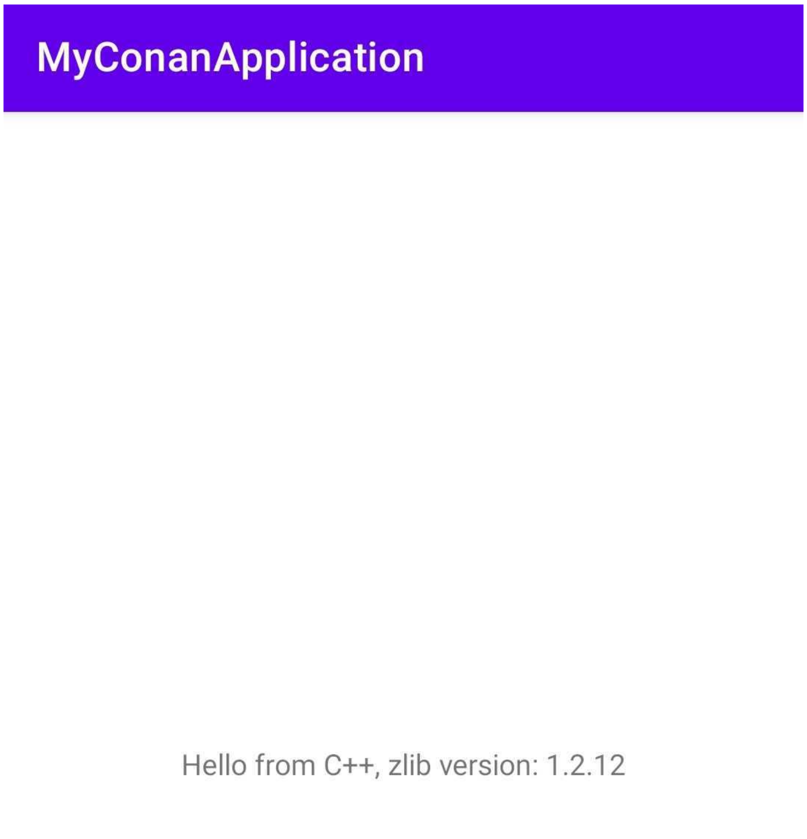
```
[requires]
zlib/1.2.12

[generators]
CMakeToolchain
CMakeDeps
```

(continues on next page)

(continued from previous page)

```
[layout]
cmake_layout
```

If we click build and then run the application, we will see that the zlib dependency has been updated:



6.5 Configuration files examples

6.5.1 Customize your settings: create your settings_user.yml

Please, first of all, 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 examples2/examples/config_files/settings_user
```

In this example we are going to see how to customize your settings without overwriting the original **settings.yml** file.

Note: To understand better this example, it is highly recommended to read previously the reference about *settings.yml*.

Locate the settings_user.yml

First of all, let's have a look at the proposed `source/settings_user.yml`:

Listing 16: **settings_user.yml**

```
os:
    webOS:
        sdk_version: [null, "7.0.0", "6.0.1", "6.0.0"]
arch: ["cortexa15t2hf"]
compiler:
    gcc:
        version: ["13.0-rc"]
```

As you can see, we don't have to rewrite all the settings because they will be merged with the already defined in **settings.yml**.

Then, what are we adding through that `settings_user.yml` file?

- New OS: `webOS`, and its sub-setting: `sdk_version`.
- New `arch` available: `cortexa15t2hf`.
- New gcc version: `13.0-rc`.

Now, it's time to copy the file `source/settings_user.yml` into your `[CONAN_HOME]/` folder:

```
$ conan config install sources/settings_user.yml
Copying file settings_user.yml to /Users/myuser/.conan2/.
```

Use your new settings

After having copied the `settings_user.yml`, you should be able to use them for your recipes. Add this simple one into your local folder:

Listing 17: **conanfile.py**

```
from conan import ConanFile

class PkgConan(ConanFile):
```

(continues on next page)

(continued from previous page)

```
    name = "pkg"
    version = "1.0"
    settings = "os", "compiler", "build_type", "arch"
```

Then, create several Conan packages (not binaries, as it does not have any source file for sure) to see that it's working correctly:

Listing 18: **Using the new OS and its sub-setting**

```
$ conan create . -s os=webOS -s os.sdk_version=7.0.0
...
Profile host:
[settings]
arch=x86_64
build_type=Release
compiler=apple-clang
compiler.cppstd=gnu98
compiler.libcxx=libc++
compiler.version=12.0
os=webOS
os.sdk_version=7.0.0

Profile build:
[settings]
arch=x86_64
build_type=Release
compiler=apple-clang
compiler.cppstd=gnu98
compiler.libcxx=libc++
compiler.version=12.0
os=Macos
...
-------- Installing (downloading, building) binaries... --------
pkg/1.0: Copying sources to build folder
pkg/1.0: Building your package in /Users/myuser/.conan2/p/t/pkg929d53a5f06b1/b
pkg/1.0: Aggregating env generators
pkg/1.0: Package 'a0d37d10fdb83a0414d7f4a1fb73da2c210211c6' built
pkg/1.0: Build folder /Users/myuser/.conan2/p/t/pkg929d53a5f06b1/b
pkg/1.0: Generated conaninfo.txt
pkg/1.0: Generating the package
pkg/1.0: Temporary package folder /Users/myuser/.conan2/p/t/pkg929d53a5f06b1/p
pkg/1.0 package(): WARN: No files in this package!
pkg/1.0: Package 'a0d37d10fdb83a0414d7f4a1fb73da2c210211c6' created
pkg/1.0: Created package revision 6a947a7b5669d6fde1a35ce5ff987fc6
pkg/1.0: Full package reference: pkg/1.0
↪#637fc1c7080faaa7e2cdccde1bcde118:a0d37d10fdb83a0414d7f4a1fb73da2c210211c6
↪#6a947a7b5669d6fde1a35ce5ff987fc6
pkg/1.0: Package folder /Users/myuser/.conan2/p/pkgb3950b1043542/p
```

Listing 19: **Using new gcc compiler version**

```
$ conan create . -s compiler=gcc -s compiler.version=13.0-rc -s compiler.
↪libcxx=libstdc++11
...
Profile host:
[settings]
```

(continues on next page)

(continued from previous page)

```
arch=x86_64
build_type=Release
compiler=gcc
compiler.libcxx=libstdc++11
compiler.version=13.0-rc
os=Macos

Profile build:
[settings]
arch=x86_64
build_type=Release
compiler=apple-clang
compiler.cppstd=gnu98
compiler.libcxx=libc++
compiler.version=12.0
os=Macos
...
-------- Installing (downloading, building) binaries... --------
pkg/1.0: Copying sources to build folder
pkg/1.0: Building your package in /Users/myuser/.conan2/p/t/pkg918904bbca9dc/b
pkg/1.0: Aggregating env generators
pkg/1.0: Package '44a4588d3fe63ccc6e7480565d35be38d405718e' built
pkg/1.0: Build folder /Users/myuser/.conan2/p/t/pkg918904bbca9dc/b
pkg/1.0: Generated conaninfo.txt
pkg/1.0: Generating the package
pkg/1.0: Temporary package folder /Users/myuser/.conan2/p/t/pkg918904bbca9dc/p
pkg/1.0 package(): WARN: No files in this package!
pkg/1.0: Package '44a4588d3fe63ccc6e7480565d35be38d405718e' created
pkg/1.0: Created package revision d913ec060e71cc56b10768afb9620094
pkg/1.0: Full package reference: pkg/1.0
↪#637fc1c7080faaa7e2cdccde1bcde118:44a4588d3fe63ccc6e7480565d35be38d405718e
↪#d913ec060e71cc56b10768afb9620094
pkg/1.0: Package folder /Users/myuser/.conan2/p/pkg789b624c93fc0/p
```

Listing 20: **Using the new OS and the new architecture**

```
$ conan create . -s os=webOS -s arch=cortexa15t2hf
...
Profile host:
[settings]
arch=cortexa15t2hf
build_type=Release
compiler=apple-clang
compiler.cppstd=gnu98
compiler.libcxx=libc++
compiler.version=12.0
os=webOS

Profile build:
[settings]
arch=x86_64
build_type=Release
compiler=apple-clang
compiler.cppstd=gnu98
compiler.libcxx=libc++
compiler.version=12.0
```

(continues on next page)

(continued from previous page)

```
os=Macos
...
-------- Installing (downloading, building) binaries... --------
pkg/1.0: Copying sources to build folder
pkg/1.0: Building your package in /Users/myuser/.conan2/p/t/pkgde9b63a6bed0a/b
pkg/1.0: Aggregating env generators
pkg/1.0: Package '19cf3cb5842b18dc78e5b0c574c1e71e7b0e17fc' built
pkg/1.0: Build folder /Users/myuser/.conan2/p/t/pkgde9b63a6bed0a/b
pkg/1.0: Generated conaninfo.txt
pkg/1.0: Generating the package
pkg/1.0: Temporary package folder /Users/myuser/.conan2/p/t/pkgde9b63a6bed0a/p
pkg/1.0 package(): WARN: No files in this package!
pkg/1.0: Package '19cf3cb5842b18dc78e5b0c574c1e71e7b0e17fc' created
pkg/1.0: Created package revision f5739d5a25b3757254dead01b30d3af0
pkg/1.0: Full package reference: pkg/1.0
↪#637fc1c7080faaa7e2cdccde1bcde118:19cf3cb5842b18dc78e5b0c574c1e71e7b0e17fc
↪#f5739d5a25b3757254dead01b30d3af0
pkg/1.0: Package folder /Users/myuser/.conan2/p/pkgd154182aac59e/p
```

As you could observe, each command has created a different package. That was completely right because we were using different settings for each one. If you want to see all the packages created, you can use the *conan list* command:

Listing 21: List all the *pkg/1.0*'s packages

```
$ conan list pkg/1.0:*
Local Cache
  pkg
    pkg/1.0
      revisions
        637fc1c7080faaa7e2cdccde1bcde118 (2023-02-16 06:42:10 UTC)
          packages
            19cf3cb5842b18dc78e5b0c574c1e71e7b0e17fc
              info
                settings
                  arch: cortexa15t2hf
                  build_type: Release
                  compiler: apple-clang
                  compiler.cppstd: gnu98
                  compiler.libcxx: libc++
                  compiler.version: 12.0
                  os: webOS
            44a4588d3fe63ccc6e7480565d35be38d405718e
              info
                settings
                  arch: x86_64
                  build_type: Release
                  compiler: gcc
                  compiler.libcxx: libstdc++11
                  compiler.version: 13.0-rc
                  os: Macos
            a0d37d10fdb83a0414d7f4a1fb73da2c210211c6
              info
                settings
                  arch: x86_64
                  build_type: Release
                  compiler: apple-clang
```

(continues on next page)

(continued from previous page)

```
                    compiler.cppstd: gnu98
                    compiler.libcxx: libc++
                    compiler.version: 12.0
                    os: webOS
                    os.sdk_version: 7.0.0
```

Try any other custom setting!

See also:

- *profiles*.
- *Conan packages binary compatibility: the package ID*

CHAPTER

SEVEN

REFERENCE

7.1 Commands

This section describe the Conan built-in commands, like `conan install` or `conan search`.

It is also possible to create user custom commands, visit *custom commands reference* and these *custom command examples*

Consumer commands:

7.1.1 conan cache

Perform file operations in the local cache (of recipes and/or packages).

conan cache path

```
$ conan cache path --help
usage: conan cache path [-h] [-v [V]] [--folder {export_source,source,build}]␣
↪reference

Show the path to the Conan cache for a given reference.

positional arguments:
    reference             Recipe reference or Package reference

optional arguments:
    -h, --help            show this help message and exit
    -v [V]                Level of detail of the output. Valid options from less
                        verbose to more verbose: -vquiet, -verror, -vwarning,
                        -vnotice, -vstatus, -v or -vverbose, -vv or -vdebug,
                        -vvv or -vtrace
    --folder {export_source,source,build}
                        Path to show. The 'build' requires a package
                        reference. If not specified it shows 'exports' path
```

The `conan cache path` returns the path in the cache of a given reference. Depending on the reference, it could return the path of a recipe, or the path to a package binary.

Let's say that we have created a package in our current cache with:

```
$ conan new cmake_lib -d name=pkg -d version=0.1
$ conan create .
...
Requirements
    pkg/0.1#cdc0d9d0e8f554d3df2388c535137d77 - Cache

Requirements
    pkg/0.1#cdc0d9d0e8f554d3df2388c535137d77:2401fa1d188d289bb25c37cfa3317e13e377a351␣
↪- Build
```

And now we are interested in obtaining the path where our `pkg/0.1` recipe `conanfile.py` has been exported:

```
$ conan cache path pkg/0.1
<path to conan cache>/p/5cb229164ec1d245/e

$ ls <path to conan cache>/p/5cb229164ec1d245/e
conanfile.py  conanmanifest.txt
```

By default, if the recipe revision is not specified, it means the "latest" revision in the cache. This can also be made explicit by the literal `#latest`, and also any recipe revision can be explicitly defined, these commands are equivalent to the above:

```
$ conan cache path pkg/0.1#latest
<path to conan cache>/p/5cb229164ec1d245/e

# The recipe revision might be different in your case.
# Check the "conan create" output to get yours
$ conan cache path pkg/0.1#cdc0d9d0e8f554d3df2388c535137d77
<path to conan cache>/p/5cb229164ec1d245/e
```

Together with the recipe folder, there are a two other folders that are common to all the binaries produced with this recipe: the "export_source" folder and the "source" folder. Both can be obtained with:

```
$ conan cache path pkg/0.1 --folder=export_source
<path to conan cache>/p/5cb229164ec1d245/es

$ ls <path to conan cache>/p/5cb229164ec1d245/es
CMakeLists.txt  include/  src/

$ conan cache path pkg/0.1 --folder=source
<path to conan cache>/p/5cb229164ec1d245/s

$ ls <path to conan cache>/p/5cb229164ec1d245/s
CMakeLists.txt  include/  src/
```

In this case the contents of the "source" folder are identical to the ones of the "export_source" folder because the recipe did not implement any `source()` method that could retrieve code or do any other operation over the code, like applying patches.

The recipe revision by default will be `#latest`, this follows the same rules as above.

Note that these two folders will not exist if the package has not been built from source, like when a precompiled binary is retrieve from a server.

It is also possible to obtain the folders of the binary packages providing the `package_id`:

```
# Your package_id might be different, it depends on the platform
# Check the "conan create" output to obtain yours
```

(continues on next page)

(continued from previous page)

```
$ conan cache path pkg/0.1:2401fa1d188d289bb25c37cfa3317e13e377a351
<path to conan cache>/p/1cae77d6250c23b7/p

$ ls <path to conan cache>/p/1cae77d6250c23b7/p
conaninfo.txt  conanmanifest.txt  include/  lib/
```

As above, by default it will resolve to the "latest" recipe revision and package revision. The command above is equal to explicitly defining `#latest` or the exact revisions. All the commands below are equivalent to the above one:

```
$ conan cache path pkg/0.1#latest:2401fa1d188d289bb25c37cfa3317e13e377a351
<path to conan cache>/p/1cae77d6250c23b7/p

$ conan cache path pkg/0.1#latest:2401fa1d188d289bb25c37cfa3317e13e377a351#latest
<path to conan cache>/p/1cae77d6250c23b7/p

$ conan cache path pkg/0.1
↪#cdc0d9d0e8f554d3df2388c535137d77:2401fa1d188d289bb25c37cfa3317e13e377a351
<path to conan cache>/p/1cae77d6250c23b7/p
```

It is possible to access the "build" folder with all the temporary build artifacts:

```
$ conan cache path pkg/0.1:2401fa1d188d289bb25c37cfa3317e13e377a351 --folder=build
<path to conan cache>/p/1cae77d6250c23b7/b

ls -al <path to conan cache>/p/1cae77d6250c23b7/b
build/  CMakeLists.txt  CMakeUserPresets.json  conaninfo.txt  include/  src/
```

Again, the "build" folder will only exist if the package was built from source.

Note: Best practices

- This `conan cache path` command is intended for eventual inspection of the cache, but the cache package storage must be considered **read-only**. Do not modify, change, remove or add files from the cache.
- If you are using this command to obtain the path to artifacts and then copying them, consider the usage of a `deployer` instead. In the general case, extracting artifacts from the cache manually is discouraged.

conan cache clean

```
$ conan cache clean -h
usage: conan cache clean [-h] [-v [V]] [-s] [-b] [-d]
                         [-p PACKAGE_QUERY]
                         pattern

Remove non-critical folders from the cache, like source, build and/or download
(.tgz store) ones.

positional arguments:
  pattern               Selection pattern for references to clean

optional arguments:
  -h, --help            show this help message and exit
  -v [V]                Level of detail of the output. Valid options from less
                        verbose to more verbose: -vquiet, -verror, -vwarning,
```

(continues on next page)

(continued from previous page)

```
                        -vnotice, -vstatus, -v or -vverbose, -vv or -vdebug,
                        -vvv or -vtrace

  -s, --source          Clean source folders
  -b, --build           Clean build folders
  -d, --download        Clean download folders
  -p PACKAGE_QUERY, --package-query PACKAGE_QUERY
                        Remove only the packages matching a specific query,
                        e.g., os=Windows AND (arch=x86 OR compiler=gcc)
```

7.1.2 conan config

Manage the Conan configuration in the Conan home.

conan config home

```
$ conan config home --help
usage: conan config home [-h] [-v [V]]
Show the Conan home folder.

optional arguments:
  -h, --help  show this help message and exit
  -v [V]      Level of detail of the output. Valid options from less verbose
              to more verbose: -vquiet, -verror, -vwarning, -vnotice,
              -vstatus, -v or -vverbose, -vv or -vdebug, -vvv or -vtrace
```

The `conan config home` command returns the path of the Conan home folder.

```
$ conan config home
```

conan config install

```
$ conan config install -h
usage: conan config install [-h] [-v [V]]
                            [--verify-ssl [VERIFY_SSL] | --insecure]
                            [-t {git,dir,file,url}] [-a ARGS]
                            [-sf SOURCE_FOLDER] [-tf TARGET_FOLDER]
                            item

Install the configuration (remotes, profiles, conf), from git, http or a
folder, into the Conan home folder.

positional arguments:
  item                  git repository, local file or folder or zip file
                        (local or http) where the configuration is stored

optional arguments:
  -h, --help            show this help message and exit
  -v [V]                Level of detail of the output. Valid options from less
                        verbose to more verbose: -vquiet, -verror, -vwarning,
                        -vnotice, -vstatus, -v or -vverbose, -vv or -vdebug,
```

(continues on next page)

(continued from previous page)

```
                        -vvv or -vtrace
  --verify-ssl [VERIFY_SSL]
                        Verify SSL connection when downloading file
  --insecure            Allow insecure server connections when using SSL.
                        Equivalent to --verify-ssl=False
  -t {git,dir,file,url}, --type {git,dir,file,url}
                        Type of remote config
  -a ARGS, --args ARGS  String with extra arguments for "git clone"
  -sf SOURCE_FOLDER, --source-folder SOURCE_FOLDER
                        Install files only from a source subfolder from the
                        specified origin
  -tf TARGET_FOLDER, --target-folder TARGET_FOLDER
                        Install to that path in the conan cache
```

The `conan config install` command is intended to install in the current home a common shared Conan configuration, like the definitions of `remotes`, `profiles`, `settings`, `hooks`, `extensions`, etc.

The command can use as source any of the following:

- A URL pointing to a zip archive containing the configuration files
- A git repository containing the files
- A local folder
- Just one file

Files in the current Conan home will be replaced by the ones from the installation source. All the configuration files can be shared and installed this way:

- `remotes.json` for the definition of remotes
- Any custom profile files inside a `profiles` subfolder
- Custom `settings.yml`
- Custom `global.conf`
- All the extensions, including plugins, hooks.
- Custom user commands.

This command reads a `.conanignore` file which, if present, filters which files and folders are copied over to the user's Conan home folder. This file uses fnmatch patterns to match over the folder contents, excluding those entries that match from the config installation. See conan-io/command-extensions's .conanignore for an example of such a file.

Examples:

- Install the configuration from a URL:

```
$ conan config install http://url/to/some/config.zip
```

- Install the configuration from a URL, but only getting the files inside a *origin* folder inside the zip file, and putting them inside a *target* folder in the local cache:

```
$ conan config install http://url/to/some/config.zip -sf=origin -tf=target
```

- Install configuration from 2 different zip files from 2 different urls, using different source and target folders for each one, then update all:

```
$ conan config install http://url/to/some/config.zip -sf=origin -tf=target
$ conan config install http://url/to/some/config.zip -sf=origin2 -tf=target2
$ conan config install http://other/url/to/other.zip -sf=hooks -tf=hooks
```

- Install the configuration from a Git repository with submodules:

```
$ conan config install http://github.com/user/conan_config/.git --args "--
↪recursive"
```

You can also force the git download by using **--type git** (in case it is not deduced from the URL automatically):

```
$ conan config install http://github.com/user/conan_config/.git --type git
```

- Install from a 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.

- Install a specific file from a local path:

```
$ conan config install my_settings/settings.yml
```

- Install the configuration from a local path:

```
$ conan config install /path/to/some/config.zip
```

conan config list

```
$ conan config list -h
usage: conan config list [-h] [-f FORMAT] [-v [V]]

Show all the Conan available configurations: core and tools.

optional arguments:
  -h, --help            show this help message and exit
  -f FORMAT, --format FORMAT
                        Select the output format: json
  -v [V]                Level of detail of the output. Valid options from less
                        verbose to more verbose: -vquiet, -verror, -vwarning,
                        -vnotice, -vstatus, -v or -vverbose, -vv or -vdebug,
                        -vvv or -vtrace
```

Displays all the Conan built-in configurations. There are 2 groups:

- `core.xxxx`: These can only be defined in `global.conf` and are used by Conan internally
- `tools.xxxx`: Thesa can be defined both in `global.conf` and profiles, and will be used by recipes and tools used within recipes, like `CMakeToolchain`

```
$ conan config list
core.cache:storage_path: Absolute path where the packages and database are stored
core.download:download_cache: Define path to a file download cache
core.download:parallel: Number of concurrent threads to download packages
core.download:retry: Number of retries in case of failure when downloading from Conan␣
↪server
```

(continues on next page)

(continued from previous page)

```
core.download:retry_wait: Seconds to wait between download attempts from Conan server
core.gzip:compresslevel: The Gzip compresion level for Conan artifacts (default=9)
core.net.http:cacert_path: Path containing a custom Cacert file
core.net.http:clean_system_proxy: If defined, the proxies system env-vars will be␣
↪discarded
core.net.http:client_cert: Path or tuple of files containing a client cert (and key)
core.net.http:max_retries: Maximum number of connection retries (requests library)
core.net.http:no_proxy_match: List of urls to skip from proxies configuration
core.net.http:proxies: Dictionary containing the proxy configuration
core.net.http:timeout: Number of seconds without response to timeout (requests␣
↪library)
core.package_id:default_build_mode: By default, 'None'
core.package_id:default_embed_mode: By default, 'full_mode'
core.package_id:default_non_embed_mode: By default, 'minor_mode'
core.package_id:default_python_mode: By default, 'minor_mode'
core.package_id:default_unknown_mode: By default, 'semver_mode'
core.upload:retry: Number of retries in case of failure when uploading to Conan server
core.upload:retry_wait: Seconds to wait between upload attempts to Conan server
core:default_build_profile: Defines the default build profile (None by default)
core:default_profile: Defines the default host profile ('default' by default)
core:non_interactive: Disable interactive user input, raises error if input necessary
core:required_conan_version: Raise if current version does not match the defined␣
↪range.
tools.android:ndk_path: Argument for the CMAKE_ANDROID_NDK
tools.apple.xcodebuild:verbosity: Verbosity level for xcodebuild: 'verbose' or 'quiet
tools.apple:enable_arc: (boolean) Enable/Disable ARC Apple Clang flags
tools.apple:enable_bitcode: (boolean) Enable/Disable Bitcode Apple Clang flags
tools.apple:enable_visibility: (boolean) Enable/Disable Visibility Apple Clang flags
tools.apple:sdk_path: Path to the SDK to be used
tools.build.cross_building:can_run: Bool value that indicates whether is possible to␣
↪run a non-native app on the same architecture. It's used by 'can_run' tool
tools.build:cflags: List of extra C flags used by different toolchains like␣
↪CMakeToolchain, AutotoolsToolchain and MesonToolchain
tools.build:compiler_executables: Defines a Python dict-like with the compilers path␣
↪to be used. Allowed keys {'c', 'cpp', 'cuda', 'objc', 'objcxx', 'rc', 'fortran',
↪'asm', 'hip', 'ispc'}
tools.build:cxxflags: List of extra CXX flags used by different toolchains like␣
↪CMakeToolchain, AutotoolsToolchain and MesonToolchain
tools.build:defines: List of extra definition flags used by different toolchains like␣
↪CMakeToolchain and AutotoolsToolchain
tools.build:download_source: Force download of sources for every package
tools.build:exelinkflags: List of extra flags used by CMakeToolchain for CMAKE_EXE_
↪LINKER_FLAGS_INIT variable
tools.build:jobs: Default compile jobs number -jX Ninja, Make, /MP VS (default: max␣
↪CPUs)
tools.build:linker_scripts: List of linker script files to pass to the linker used by␣
↪different toolchains like CMakeToolchain, AutotoolsToolchain, and MesonToolchain
tools.build:sharedlinkflags: List of extra flags used by CMakeToolchain for CMAKE_
↪SHARED_LINKER_FLAGS_INIT variable
tools.build:skip_test: Do not execute CMake.test() and Meson.test() when enabled
tools.build:sysroot: Pass the --sysroot=<tools.build:sysroot> flag if available.␣
↪(None by default)
tools.cmake.cmake_layout:build_folder_vars: Settings and Options that will produce a␣
↪different build folder and different CMake presets names
tools.cmake.cmaketoolchain:find_package_prefer_config: Argument for the CMAKE_FIND_
↪PACKAGE_PREFER_CONFIG
tools.cmake.cmaketoolchain:generator: User defined CMake generator to use instead of␣
↪default
```

(continues on next page)

(continued from previous page)

```
tools.cmake.cmaketoolchain:system_name: Define CMAKE_SYSTEM_NAME in CMakeToolchain
tools.cmake.cmaketoolchain:system_processor: Define CMAKE_SYSTEM_PROCESSOR in␣
↪CMakeToolchain
tools.cmake.cmaketoolchain:system_version: Define CMAKE_SYSTEM_VERSION in␣
↪CMakeToolchain
tools.cmake.cmaketoolchain:toolchain_file: Use other existing file rather than conan_
↪toolchain.cmake one
tools.cmake.cmaketoolchain:toolset_arch: Toolset architecture to be used as part of␣
↪CMAKE_GENERATOR_TOOLSET in CMakeToolchain
tools.cmake.cmaketoolchain:user_toolchain: Inject existing user toolchains at the␣
↪beginning of conan_toolchain.cmake
tools.env.virtualenv:powershell: If it is set to True it will generate powershell␣
↪launchers if os=Windows
tools.files.download:download_cache: Define the cache folder to store downloads from␣
↪files.download()/get()
tools.files.download:retry: Number of retries in case of failure when downloading
tools.files.download:retry_wait: Seconds to wait between download attempts
tools.gnu:define_libcxx11_abi: Force definition of GLIBCXX_USE_CXX11_ABI=1 for␣
↪libstdc++11
tools.gnu:host_triplet: Custom host triplet to pass to Autotools scripts
tools.gnu:make_program: Indicate path to make program
tools.gnu:pkg_config: Path to pkg-config executable used by PkgConfig build helper
tools.google.bazel:bazelrc_path: Defines Bazel rc-path
tools.google.bazel:configs: Define Bazel config file
tools.info.package_id:confs: List of existing configuration to be part of the package␣
↪ID
tools.intel:installation_path: Defines the Intel oneAPI installation root path
tools.intel:setvars_args: Custom arguments to be passed onto the setvars.sh|bat␣
↪script from Intel oneAPI
tools.meson.mesontoolchain:backend: Any Meson backend: ninja, vs, vs2010, vs2012,␣
↪vs2013, vs2015, vs2017, vs2019, xcode
tools.meson.mesontoolchain:extra_machine_files: List of paths for any additional␣
↪native/cross file references to be appended to the existing Conan ones
tools.microsoft.bash:active: If Conan is already running inside bash terminal in␣
↪Windows
tools.microsoft.bash:path: The path to the shell to run when conanfile.win_bash==True
tools.microsoft.bash:subsystem: The subsystem to be used when conanfile.win_
↪bash==True. Possible values: msys2, msys, cygwin, wsl, sfu
tools.microsoft.msbuild:installation_path: VS install path, to avoid auto-detect via␣
↪vswhere, like C:/Program Files (x86)/Microsoft Visual Studio/2019/Community. Use␣
↪empty string to disable
tools.microsoft.msbuild:max_cpu_count: Argument for the /m when running msvc to build␣
↪parallel projects
tools.microsoft.msbuild:verbosity: Verbosity level for MSBuild: 'Quiet', 'Minimal',
↪'Normal', 'Detailed', 'Diagnostic'
tools.microsoft.msbuild:vs_version: Defines the IDE version when using the new msvc␣
↪compiler
tools.microsoft.msbuilddeps:exclude_code_analysis: Suppress MSBuild code analysis for␣
↪patterns
tools.microsoft.msbuildtoolchain:compile_options: Dictionary with MSBuild compiler␣
↪options
tools.system.package_manager:mode: Mode for package_manager tools: 'check' or 'install
↪'
tools.system.package_manager:sudo: Use 'sudo' when invoking the package manager tools␣
↪in Linux (False by default)
tools.system.package_manager:sudo_askpass: Use the '-A' argument if using sudo in␣
↪Linux to invoke the system package manager (False by default)
```

(continues on next page)

(continued from previous page)

```
tools.system.package_manager:tool: Default package manager tool: 'apt-get', 'yum',
↪'dnf', 'brew', 'pacman', 'choco', 'zypper', 'pkg' or 'pkgutil'
```

See also:

- *Conan configuration files*

7.1.3 conan graph

The `conan graph` command contains several subcommands that return information of a dependency graph without needing to download the package binaries.

conan graph info

```
$ conan graph info -h
usage: conan graph info [-h] [-f FORMAT] [-v [V]] [--name NAME]
                        [--version VERSION] [--user USER] [--channel CHANNEL]
                        [--requires REQUIRES] [--tool-requires TOOL_REQUIRES]
                        [-b BUILD] [-r REMOTE | -nr] [-u] [-o OPTIONS_HOST]
                        [-o:b OPTIONS_BUILD] [-o:h OPTIONS_HOST]
                        [-pr PROFILE_HOST] [-pr:b PROFILE_BUILD]
                        [-pr:h PROFILE_HOST] [-s SETTINGS_HOST]
                        [-s:b SETTINGS_BUILD] [-s:h SETTINGS_HOST]
                        [-c CONF_HOST] [-c:b CONF_BUILD] [-c:h CONF_HOST]
                        [-l LOCKFILE] [--lockfile-partial]
                        [--lockfile-out LOCKFILE_OUT] [--lockfile-packages]
                        [--lockfile-clean] [--check-updates] [--filter FILTER]
                        [--package-filter PACKAGE_FILTER] [--deploy DEPLOY]
                        [path]

Compute the dependency graph and shows information about it.

positional arguments:
  path                  Path to a folder containing a recipe (conanfile.py or
                        conanfile.txt) or to a recipe file. e.g.,
                        ./my_project/conanfile.txt.

optional arguments:
  -h, --help            show this help message and exit
  -f FORMAT, --format FORMAT
                        Select the output format: html, json, dot
  -v [V]                Level of detail of the output. Valid options from less
                        verbose to more verbose: -vquiet, -verror, -vwarning,
                        -vnotice, -vstatus, -v or -vverbose, -vv or -vdebug,
                        -vvv or -vtrace
  --name NAME           Provide a package name if not specified in conanfile
  --version VERSION     Provide a package version if not specified in
                        conanfile
  --user USER           Provide a user if not specified in conanfile
  --channel CHANNEL     Provide a channel if not specified in conanfile
  --requires REQUIRES   Directly provide requires instead of a conanfile
  --tool-requires TOOL_REQUIRES
                        Directly provide tool-requires instead of a conanfile
  -b BUILD, --build BUILD
```

(continues on next page)

(continued from previous page)

```
                        Optional, specify which packages to build from source.
                        Combining multiple '--build' options on one command
                        line is allowed. Possible values: --build="*" Force
                        build from source for all packages. --build=never
                        Disallow build for all packages, use binary packages
                        or fail if a binary package is not found. Cannot be
                        combined with other '--build' options. --build=missing
                        Build packages from source whose binary package is not
                        found. --build=cascade Build packages from source that
                        have at least one dependency being built from source.
                        --build=[pattern] Build packages from source whose
                        package reference matches the pattern. The pattern
                        uses 'fnmatch' style wildcards. --build=![pattern]
                        Excluded packages, which will not be built from the
                        source, whose package reference matches the pattern.
                        The pattern uses 'fnmatch' style wildcards.
                        --build=missing:[pattern] Build from source if a
                        compatible binary does not exist, only for packages
                        matching pattern.
  -r REMOTE, --remote REMOTE
                        Look in the specified remote or remotes server
  -nr, --no-remote      Do not use remote, resolve exclusively in the cache
  -u, --update          Will check the remote and in case a newer version
                        and/or revision of the dependencies exists there, it
                        will install those in the local cache. When using
                        version ranges, it will install the latest version
                        that satisfies the range. Also, if using revisions, it
                        will update to the latest revision for the resolved
                        version range.
  -o OPTIONS_HOST, --options OPTIONS_HOST
                        Define options values (host machine), e.g.: -o
                        Pkg:with_qt=true
  -o:b OPTIONS_BUILD, --options:build OPTIONS_BUILD
                        Define options values (build machine), e.g.: -o:b
                        Pkg:with_qt=true
  -o:h OPTIONS_HOST, --options:host OPTIONS_HOST
                        Define options values (host machine), e.g.: -o:h
                        Pkg:with_qt=true
  -pr PROFILE_HOST, --profile PROFILE_HOST
                        Apply the specified profile to the host machine
  -pr:b PROFILE_BUILD, --profile:build PROFILE_BUILD
                        Apply the specified profile to the build machine
  -pr:h PROFILE_HOST, --profile:host PROFILE_HOST
                        Apply the specified profile to the host machine
  -s SETTINGS_HOST, --settings SETTINGS_HOST
                        Settings to build the package, overwriting the
                        defaults (host machine). e.g.: -s compiler=gcc
  -s:b SETTINGS_BUILD, --settings:build SETTINGS_BUILD
                        Settings to build the package, overwriting the
                        defaults (build machine). e.g.: -s:b compiler=gcc
  -s:h SETTINGS_HOST, --settings:host SETTINGS_HOST
                        Settings to build the package, overwriting the
                        defaults (host machine). e.g.: -s:h compiler=gcc
  -c CONF_HOST, --conf CONF_HOST
                        Configuration to build the package, overwriting the
                        defaults (host machine). e.g.: -c
                        tools.cmake.cmaketoolchain:generator=Xcode
```

(continues on next page)

(continued from previous page)

```
  -c:b CONF_BUILD, --conf:build CONF_BUILD
                        Configuration to build the package, overwriting the
                        defaults (build machine). e.g.: -c:b
                        tools.cmake.cmaketoolchain:generator=Xcode
  -c:h CONF_HOST, --conf:host CONF_HOST
                        Configuration to build the package, overwriting the
                        defaults (host machine). e.g.: -c:h
                        tools.cmake.cmaketoolchain:generator=Xcode
  -l LOCKFILE, --lockfile LOCKFILE
                        Path to a lockfile. Use --lockfile="" to avoid
                        automatic use of existing 'conan.lock' file
  --lockfile-partial    Do not raise an error if some dependency is not found
                        in lockfile
  --lockfile-out LOCKFILE_OUT
                        Filename of the updated lockfile
  --lockfile-packages   Lock package-id and package-revision information
  --lockfile-clean      Remove unused entries from the lockfile
  --check-updates       Check if there are recipe updates
  --filter FILTER       Show only the specified fields
  --package-filter PACKAGE_FILTER
                        Print information only for packages that match the
                        patterns
  --deploy DEPLOY       Deploy using the provided deployer to the output
                        folder
```

The `conan graph info` command shows information about the dependency graph for the recipe specified in `path`.

> **Warning:** The json output of the `conan graph --format=json` is **experimental** and subject to change.

conan graph build-order

```
$ conan graph build-order -h
usage: conan graph build-order [-h] [-f FORMAT] [-v [V]]
                               [--name NAME] [--version VERSION] [--user USER]
                               [--channel CHANNEL] [--requires REQUIRES]
                               [--tool-requires TOOL_REQUIRES] [-b BUILD]
                               [-r REMOTE | -nr] [-u] [-o OPTIONS_HOST]
                               [-o:b OPTIONS_BUILD] [-o:h OPTIONS_HOST]
                               [-pr PROFILE_HOST] [-pr:b PROFILE_BUILD]
                               [-pr:h PROFILE_HOST] [-s SETTINGS_HOST]
                               [-s:b SETTINGS_BUILD] [-s:h SETTINGS_HOST]
                               [-c CONF_HOST] [-c:b CONF_BUILD]
                               [-c:h CONF_HOST] [-l LOCKFILE]
                               [--lockfile-partial]
                               [--lockfile-out LOCKFILE_OUT]
                               [--lockfile-packages] [--lockfile-clean]
                               [path]

Compute the build order of a dependency graph.

positional arguments:
  path                  Path to a folder containing a recipe (conanfile.py or
```

(continues on next page)

(continued from previous page)

```
                        conanfile.txt) or to a recipe file. e.g.,
                        ./my_project/conanfile.txt.

optional arguments:
  -h, --help            show this help message and exit
  -f FORMAT, --format FORMAT
                        Select the output format: json
  -v [V]                Level of detail of the output. Valid options from less
                        verbose to more verbose: -vquiet, -verror, -vwarning,
                        -vnotice, -vstatus, -v or -vverbose, -vv or -vdebug,
                        -vvv or -vtrace
  --name NAME           Provide a package name if not specified in conanfile
  --version VERSION     Provide a package version if not specified in
                        conanfile
  --user USER           Provide a user if not specified in conanfile
  --channel CHANNEL     Provide a channel if not specified in conanfile
  --requires REQUIRES   Directly provide requires instead of a conanfile
  --tool-requires TOOL_REQUIRES
                        Directly provide tool-requires instead of a conanfile
  -b BUILD, --build BUILD
                        Optional, specify which packages to build from source.
                        Combining multiple '--build' options on one command
                        line is allowed. Possible values: --build="*" Force
                        build from source for all packages. --build=never
                        Disallow build for all packages, use binary packages
                        or fail if a binary package is not found. Cannot be
                        combined with other '--build' options. --build=missing
                        Build packages from source whose binary package is not
                        found. --build=cascade Build packages from source that
                        have at least one dependency being built from source.
                        --build=[pattern] Build packages from source whose
                        package reference matches the pattern. The pattern
                        uses 'fnmatch' style wildcards. --build=![pattern]
                        Excluded packages, which will not be built from the
                        source, whose package reference matches the pattern.
                        The pattern uses 'fnmatch' style wildcards.
                        --build=missing:[pattern] Build from source if a
                        compatible binary does not exist, only for packages
                        matching pattern.
  -r REMOTE, --remote REMOTE
                        Look in the specified remote or remotes server
  -nr, --no-remote      Do not use remote, resolve exclusively in the cache
  -u, --update          Will check the remote and in case a newer version
                        and/or revision of the dependencies exists there, it
                        will install those in the local cache. When using
                        version ranges, it will install the latest version
                        that satisfies the range. Also, if using revisions, it
                        will update to the latest revision for the resolved
                        version range.
  -o OPTIONS_HOST, --options OPTIONS_HOST
                        Define options values (host machine), e.g.: -o
                        Pkg:with_qt=true
  -o:b OPTIONS_BUILD, --options:build OPTIONS_BUILD
                        Define options values (build machine), e.g.: -o:b
                        Pkg:with_qt=true
  -o:h OPTIONS_HOST, --options:host OPTIONS_HOST
                        Define options values (host machine), e.g.: -o:h
```

(continues on next page)

(continued from previous page)

```
                        Pkg:with_qt=true
  -pr PROFILE_HOST, --profile PROFILE_HOST
                        Apply the specified profile to the host machine
  -pr:b PROFILE_BUILD, --profile:build PROFILE_BUILD
                        Apply the specified profile to the build machine
  -pr:h PROFILE_HOST, --profile:host PROFILE_HOST
                        Apply the specified profile to the host machine
  -s SETTINGS_HOST, --settings SETTINGS_HOST
                        Settings to build the package, overwriting the
                        defaults (host machine). e.g.: -s compiler=gcc
  -s:b SETTINGS_BUILD, --settings:build SETTINGS_BUILD
                        Settings to build the package, overwriting the
                        defaults (build machine). e.g.: -s:b compiler=gcc
  -s:h SETTINGS_HOST, --settings:host SETTINGS_HOST
                        Settings to build the package, overwriting the
                        defaults (host machine). e.g.: -s:h compiler=gcc
  -c CONF_HOST, --conf CONF_HOST
                        Configuration to build the package, overwriting the
                        defaults (host machine). e.g.: -c
                        tools.cmake.cmaketoolchain:generator=Xcode
  -c:b CONF_BUILD, --conf:build CONF_BUILD
                        Configuration to build the package, overwriting the
                        defaults (build machine). e.g.: -c:b
                        tools.cmake.cmaketoolchain:generator=Xcode
  -c:h CONF_HOST, --conf:host CONF_HOST
                        Configuration to build the package, overwriting the
                        defaults (host machine). e.g.: -c:h
                        tools.cmake.cmaketoolchain:generator=Xcode
  -l LOCKFILE, --lockfile LOCKFILE
                        Path to a lockfile. Use --lockfile="" to avoid
                        automatic use of existing 'conan.lock' file
  --lockfile-partial    Do not raise an error if some dependency is not found
                        in lockfile
  --lockfile-out LOCKFILE_OUT
                        Filename of the updated lockfile
  --lockfile-packages   Lock package-id and package-revision information
  --lockfile-clean      Remove unused entries from the lockfile
```

The `conan graph build-order` command computes build order of the dependency graph for the recipe specified in `path`.

conan graph build-order-merge

```
$ conan graph build-order-merge -h
usage: conan graph build-order-merge [-h] [-f FORMAT] [-v [V]]
                                     [--file [FILE]]

Merge more than 1 build-order file.

optional arguments:
  -h, --help            show this help message and exit
  -f FORMAT, --format FORMAT
                        Select the output format: json
  -v [V]                Level of detail of the output. Valid options from less
                        verbose to more verbose: -vquiet, -verror, -vwarning,
```

(continues on next page)

(continued from previous page)

```
                        -vnotice, -vstatus, -v or -vverbose, -vv or -vdebug,
                        -vvv or -vtrace
  --file [FILE]         Files to be merged
```

7.1.4 conan inspect

Warning: This feature is experimental and subject to breaking changes.

```
$ conan inspect -h
usage: conan inspect [-h] [-f FORMAT] [-v [V]] path

Inspect a conanfile.py to return its public fields.

positional arguments:
  path                  Path to a folder containing a recipe (conanfile.py)

optional arguments:
  -h, --help            show this help message and exit
  -f FORMAT, --format FORMAT
                        Select the output format: json
  -v [V]                Level of detail of the output. Valid options from less
                        verbose to more verbose: -vquiet, -verror, -vwarning,
                        -vnotice, -vstatus, -v or -vverbose, -vv or -vdebug,
                        -vvv or -vtrace
```

Note: `conan inspect` doesn't really evaluate any methods or apply any conditional logic. It lists class attributes only.

The **`conan inspect`** command shows the public attributes of any recipe (*conanfile.py*) as follows:

```
$ conan inspect .
default_options:
    shared: False
    fPIC: True
    neon: True
    msa: True
    sse: True
    vsx: True
    api_prefix:
description: libpng is the official PNG file format reference library.
generators: []
homepage: http://www.libpng.org
license: libpng-2.0
name: libpng
no_copy_source: False
options:
    shared: [True, False]
    fPIC: [True, False]
    neon: [True, 'check', False]
    msa: [True, False]
```

(continues on next page)

(continued from previous page)

```
    sse: [True, False]
    vsx: [True, False]
    api_prefix: ['ANY']
revision_mode: hash
settings: ('os', 'arch', 'compiler', 'build_type')
topics: ('png', 'graphics', 'image')
url: https://github.com/conan-io/conan-center-index
```

The **conan inspect ... --format=json** returns a JSON output format in `stdout` (which can be redirected to a file) with the following structure:

```
$ conan inspect -f json .
{
    "author": null,
    "build_policy": null,
    "build_requires": null,
    "buildenv_info": null,
    "channel": null,
    "conf_info": null,
    "cpp": null,
    "default_options": {
        "shared": false,
        "fPIC": true,
        "neon": true,
        "msa": true,
        "sse": true,
        "vsx": true,
        "api_prefix": ""
    },
    "deprecated": null,
    "description": "libpng is the official PNG file format reference library.",
    "exports": null,
    "exports_sources": null,
    "generators": [],
    "homepage": "http://www.libpng.org",
    "license": "libpng-2.0",
    "name": "libpng",
    "no_copy_source": false,
    "options": {
        "shared": [
            true,
            false
        ],
        "fPIC": [
            true,
            false
        ],
        "neon": [
            true,
            "check",
            false
        ],
        "msa": [
            true,
            false
        ],
```

(continues on next page)

(continued from previous page)

```
        "sse": [
            true,
            false
        ],
        "vsx": [
            true,
            false
        ],
        "api_prefix": [
            "ANY"
        ]
    },
    "package_type": null,
    "provides": null,
    "recipe_folder": null,
    "requires": null,
    "revision_mode": "hash",
    "runenv_info": null,
    "settings": [
        "os",
        "arch",
        "compiler",
        "build_type"
    ],
    "test_requires": null,
    "tested_reference_str": null,
    "tool_requires": null,
    "topics": [
        "png",
        "graphics",
        "image"
    ],
    "upload_policy": null,
    "url": "https://github.com/conan-io/conan-center-index",
    "user": null,
    "version": null,
    "win_bash": null,
    "win_bash_run": null
}
```

7.1.5 conan install

```
$ conan install -h
usage: conan install [-h] [-f FORMAT] [-v [V]] [--name NAME]
                     [--version VERSION] [--user USER] [--channel CHANNEL]
                     [--requires REQUIRES] [--tool-requires TOOL_REQUIRES]
                     [-b BUILD] [-r REMOTE | -nr] [-u] [-o OPTIONS_HOST]
                     [-o:b OPTIONS_BUILD] [-o:h OPTIONS_HOST]
                     [-pr PROFILE_HOST] [-pr:b PROFILE_BUILD]
                     [-pr:h PROFILE_HOST] [-s SETTINGS_HOST]
                     [-s:b SETTINGS_BUILD] [-s:h SETTINGS_HOST] [-c CONF_HOST]
                     [-c:b CONF_BUILD] [-c:h CONF_HOST] [-l LOCKFILE]
                     [--lockfile-partial] [--lockfile-out LOCKFILE_OUT]
                     [--lockfile-packages] [--lockfile-clean] [-g GENERATOR]
                     [-of OUTPUT_FOLDER] [--deploy DEPLOY]
```

(continues on next page)

(continued from previous page)

```
                     [path]

Install 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 built from sources using the '--build' option. When
the package is installed, Conan will write the files for the specified
generators.

positional arguments:
  path                  Path to a folder containing a recipe (conanfile.py or
                        conanfile.txt) or to a recipe file. e.g.,
                        ./my_project/conanfile.txt.

optional arguments:
  -h, --help            show this help message and exit
  -f FORMAT, --format FORMAT
                        Select the output format: json
  -v [V]                Level of detail of the output. Valid options from less
                        verbose to more verbose: -vquiet, -verror, -vwarning,
                        -vnotice, -vstatus, -v or -vverbose, -vv or -vdebug,
                        -vvv or -vtrace
  --name NAME           Provide a package name if not specified in conanfile
  --version VERSION     Provide a package version if not specified in
                        conanfile
  --user USER           Provide a user if not specified in conanfile
  --channel CHANNEL     Provide a channel if not specified in conanfile
  --requires REQUIRES   Directly provide requires instead of a conanfile
  --tool-requires TOOL_REQUIRES
                        Directly provide tool-requires instead of a conanfile
  -b BUILD, --build BUILD
                        Optional, specify which packages to build from source.
                        Combining multiple '--build' options on one command
                        line is allowed. Possible values: --build="*" Force
                        build from source for all packages. --build=never
                        Disallow build for all packages, use binary packages
                        or fail if a binary package is not found. Cannot be
                        combined with other '--build' options. --build=missing
                        Build packages from source whose binary package is not
                        found. --build=cascade Build packages from source that
                        have at least one dependency being built from source.
                        --build=[pattern] Build packages from source whose
                        package reference matches the pattern. The pattern
                        uses 'fnmatch' style wildcards. --build=![pattern]
                        Excluded packages, which will not be built from the
                        source, whose package reference matches the pattern.
                        The pattern uses 'fnmatch' style wildcards.
                        --build=missing:[pattern] Build from source if a
                        compatible binary does not exist, only for packages
                        matching pattern.
  -r REMOTE, --remote REMOTE
                        Look in the specified remote or remotes server
```

(continues on next page)

(continued from previous page)

```
  -nr, --no-remote      Do not use remote, resolve exclusively in the cache
  -u, --update          Will check the remote and in case a newer version
                        and/or revision of the dependencies exists there, it
                        will install those in the local cache. When using
                        version ranges, it will install the latest version
                        that satisfies the range. Also, if using revisions, it
                        will update to the latest revision for the resolved
                        version range.
  -o OPTIONS_HOST, --options OPTIONS_HOST
                        Define options values (host machine), e.g.: -o
                        Pkg:with_qt=true
  -o:b OPTIONS_BUILD, --options:build OPTIONS_BUILD
                        Define options values (build machine), e.g.: -o:b
                        Pkg:with_qt=true
  -o:h OPTIONS_HOST, --options:host OPTIONS_HOST
                        Define options values (host machine), e.g.: -o:h
                        Pkg:with_qt=true
  -pr PROFILE_HOST, --profile PROFILE_HOST
                        Apply the specified profile to the host machine
  -pr:b PROFILE_BUILD, --profile:build PROFILE_BUILD
                        Apply the specified profile to the build machine
  -pr:h PROFILE_HOST, --profile:host PROFILE_HOST
                        Apply the specified profile to the host machine
  -s SETTINGS_HOST, --settings SETTINGS_HOST
                        Settings to build the package, overwriting the
                        defaults (host machine). e.g.: -s compiler=gcc
  -s:b SETTINGS_BUILD, --settings:build SETTINGS_BUILD
                        Settings to build the package, overwriting the
                        defaults (build machine). e.g.: -s:b compiler=gcc
  -s:h SETTINGS_HOST, --settings:host SETTINGS_HOST
                        Settings to build the package, overwriting the
                        defaults (host machine). e.g.: -s:h compiler=gcc
  -c CONF_HOST, --conf CONF_HOST
                        Configuration to build the package, overwriting the
                        defaults (host machine). e.g.: -c
                        tools.cmake.cmaketoolchain:generator=Xcode
  -c:b CONF_BUILD, --conf:build CONF_BUILD
                        Configuration to build the package, overwriting the
                        defaults (build machine). e.g.: -c:b
                        tools.cmake.cmaketoolchain:generator=Xcode
  -c:h CONF_HOST, --conf:host CONF_HOST
                        Configuration to build the package, overwriting the
                        defaults (host machine). e.g.: -c:h
                        tools.cmake.cmaketoolchain:generator=Xcode
  -l LOCKFILE, --lockfile LOCKFILE
                        Path to a lockfile. Use --lockfile="" to avoid
                        automatic use of existing 'conan.lock' file
  --lockfile-partial    Do not raise an error if some dependency is not found
                        in lockfile
  --lockfile-out LOCKFILE_OUT
                        Filename of the updated lockfile
  --lockfile-packages   Lock package-id and package-revision information
  --lockfile-clean      Remove unused entries from the lockfile
  -g GENERATOR, --generator GENERATOR
                        Generators to use
  -of OUTPUT_FOLDER, --output-folder OUTPUT_FOLDER
                        The root output folder for generated and build files
```

(continues on next page)

(continued from previous page)

```
  --deploy DEPLOY       Deploy using the provided deployer to the output
                        folder
```

The `conan install` command is one of the main Conan commands, and it is used to resolve and install dependencies.

This command does the following:

- Compute the whole dependency graph, for the current configuration defined by settings, options, profiles and configuration. It resolves version ranges, transitive dependencies, conditional requirements, etc, to build the dependency graph.
- Evaluate the existence of binaries for every package in the graph, whether or not there are precompiled binaries to download, or if they should be built from sources (as directed by the `--build` argument). If binaries are missing, it will not recompute the dependency graph to try to fallback to previous versions that contain binaries for that configuration. If a certain dependency version is desired, it should be explicitly required.
- Download precompiled binaries, or build binaries from sources in the local cache, in the right order for the dependency graph.
- Create the necessary files as requested by the "generators", so build systems and other tools can locate the locally installed dependencies
- Optionally, execute the desired `deployers`.

Warning: The json output of the `conan install --format=json` is **experimental** and subject to change.

Conanfile path or –requires

The `conan install` command can use 2 different origins for information. The first one is using a local `conanfile.py` or `conanfile.txt`, containing definitions of the dependencies and generators to be used.

```
$ conan install .  # there is a conanfile.txt or a conanfile.py in the cwd
$ conan install conanfile.py  # also works, direct reference file
$ conan install myconan.txt  # explicit custom name
$ conan install myfolder  # there is a conanfile in "myfolder" folder
```

Even if it is possible to use a custom name, in the general case, it is recommended to use the default `conanfile.py` name, located in the repository root, so users can do a straightforward `git clone ... `` + ``conan install .`

The other possibility is to not have a `conanfile` at all, and define the requirements to be installed directly in the command line:

```
# Install the zlib/1.2.13 library
$ conan install --requires=zlib/1.2.13
# Install the zlib/1.2.13 and bzip2/1.0.8 libraries
$ conan install --requires=zlib/1.2.13 --requires=bzip2/1.0.8
# Install the cmake/3.23.5 and ninja/1.11.0 tools
$ conan install --tool-requires=cmake/3.23.5 --tool-requires=ninja/1.11.0
# Install the zlib/1.2.13 library and ninja/1.11.0 tool
$ conan install --requires=zlib/1.2.13 --tool-requires=ninja/1.11.0
```

In the general case, it is recommended to use a `conanfile` instead of defining things in the command line.

Profiles, Settings, Options, Conf

There are several arguments that are used to define the effective profiles that will be used, both for the "build" and "host" contexts.

By default the arguments refer to the "host" context, so `--settings:host, -s:h` is totally equivalent to `--settings, -s`. Also, by default, the `conan install` command will use the `default` profile both for the "build" and "host" context. That means that if a profile with the "default" name has not been created, it will error.

Multiple definitions of profiles can be passed as arguments, and they will compound from left to right (right has the highest priority)

```
# The values of myprofile3 will have higher priority
$ conan install . -pr=myprofile1 -pr=myprofile2 -pr=myprofile3
```

If values for any of `settings`, `options` and `conf` are provided in the command line, they create a profile that is composed with the other provided `-pr` (or the "default" one if not specified) profiles, with higher priority, not matter what the order of arguments is.

```
# the final "host" profile will always be build_type=Debug, even if "myprofile"
# says "build_type=Release"
$ conan install . -pr=myprofile -s build_type=Debug
```

Generators and deployers

The `-g` argument allows to define in the command line the different built-in generators to be used:

```
$ conan install --requires=zlib/1.2.13 -g CMakeDeps -g CMakeToolchain
```

Note that in the general case, the recommended approach is to have the `generators` defined in the `conanfile`, and only for the `--requires` use case, it would be more necessary as command line argument.

Generators are intended to create files for the build systems to locate the dependencies, while the `deployers` main use case is to copy files from the Conan cache to user space, and performing any other custom operations over the dependency graph, like collecting licenses, generating reports, deploying binaries to the system, etc. The syntax for deployers is:

```
# does a full copy of the dependencies binaries to the current user folder
$ conan install . --deploy=full_deploy
```

There are 2 built-in deployers:

- `full_deploy` does a complete copy of the dependencies binaries in the local folder, with a minimal folder structure to avoid conflicts between files and artifacts of different packages
- `direct_deploy` does a copy of only the immediate direct dependencies, but does not include the transitive dependencies.

Some generators might have the capability of redefining the target "package folder". That means that if some other generator like `CMakeDeps` is used that is pointing to the packages, it will be pointing to the local deployed copy, and not to the original packages in the Conan cache.

It is also possible, and it is a powerful extension point, to write custom user deployers. Read more about custom deployers in *Deployers*.

Name, version, user, channel

The `conan install` command provides optional arguments for `--name`, `--version`, `--user`, `--channel`. These arguments might not be necessary in the majority of cases. Never for `conanfile.txt` and for `conanfile.py` only in the case that they are not defined in the recipe:

```
from conan import ConanFile
from conan.tools.scm import Version

class Pkg(ConanFile):
    name = "mypkg"

    def requirements(self):
        if Version(self.version) >= "3.23":
            self.requires("...")
```

```
# If we don't specify ``--version``, it will be None and it will fail
$ conan install . --version=3.24
```

Lockfiles

The `conan install` command has several arguments to load and produce lockfiles. By default, if a `conan.lock` file is located beside the recipe or in the current working directory if no path is provided, will be used as an input lockfile.

Lockfiles are strict by default, that means that if there is some `requires` and it cannot find a matching locked reference in the lockfile, it will error and stop. For cases where it is expected that the lockfile will not be complete, as there might be new dependencies, the `--lockfile-partial` argument can be used.

By default, `conan install` will not generate an output lockfile, but if the `--lockfile-out` argument is provided, pointing to a filename, like `--lockfile-out=result.lock`, then a lockfile will be generated from the current dependency graph. If `--lockfile-clean` argument is provided, all versions and revisions not used in the current dependency graph will be dropped from the resulting lockfile.

Let's say that we already have a `conan.lock` input lockfile, but we just added a new `requires = "newpkg/1.0"` to a new dependency. We could resolve the dependencies, locking all the previously locked versions, while allowing to resolve the new one, which was not previously present in the lockfile, and store it in a new location, or overwrite the existing lockfile:

```
# --lockfile=conan.lock is the default, not necessary
$ conan install . --lockfile=conan.lock --lockfile-partial --lockfile-out=conan.lock
```

The `--lockfile-packages` argument allows to create lockfiles that also lock down to the package revision, but it should not be necessary in the vast majority of cases, so it is discouraged in the general case.

Also, it is likely that the majority of lockfile operations are better managed by the `conan lock` command.

Read more about lockfiles in *Lockfiles*.

See also:

- Read the tutorial about the *local package developement flow*.

7.1.6 conan list

```
$ conan list -h
usage: conan list [-h] [-f FORMAT] [-v [V]] [-p PACKAGE_QUERY]
                  [-r REMOTE] [-c]
                  reference

List existing recipes, revisions, or packages in the cache (by default) or the␣
↪remotes.

positional arguments:
  reference             Recipe reference or package reference. Both can
                        contain * as wildcard at any reference field. If
                        revision is not specified, it is assumed latest one.

optional arguments:
  -h, --help            show this help message and exit
  -f FORMAT, --format FORMAT
                        Select the output format: json, html
  -v [V]                Level of detail of the output. Valid options from less
                        verbose to more verbose: -vquiet, -verror, -vwarning,
                        -vnotice, -vstatus, -v or -vverbose, -vv or -vdebug,
                        -vvv or -vtrace
  -p PACKAGE_QUERY, --package-query PACKAGE_QUERY
                        List only the packages matching a specific query, e.g,
                        os=Windows AND (arch=x86 OR compiler=gcc)
  -r REMOTE, --remote REMOTE
                        Remote names. Accepts wildcards ('*' means all the
                        remotes available)
  -c, --cache           Search in the local cache
```

The `conan list` command can list recipes and packages from the local cache, from the specified remotes or from both. This command uses a *reference pattern* as input. The structure of this pattern is based on a complete Conan reference that looks like:

`name/version@user/channel#rrev:pkgid#prev`

This pattern supports using `*` as wildcard as well as `#latest` to specify the latest revision (though that might not be necessary in most cases, by default Conan will be listing the latest revisions).

Using it you can list:

- Recipe references (`name/version@user/channel`).
- Recipe revisions (`name/version@user/channel#rrev`).
- Package IDs and their configurations (`name/version@user/channel#rrev:pkgids`).
- Package revisions (`name/version@user/channel#rrev:pkgids#prev`).

> **Warning:** The json output of the `conan list --format=json` is in **preview**. See *the Conan stability* section for more information.

Let's see some examples on how to use this pattern:

Listing recipe references

Listing 1: *list all references on local cache*

```
$ conan list *
Local Cache
  hello
    hello/2.26.1@mycompany/testing
    hello/2.20.2@mycompany/testing
    hello/1.0.4@mycompany/testing
    hello/2.3.2@mycompany/stable
    hello/1.0.4@mycompany/stable
  string-view-lite
    string-view-lite/1.6.0
  zlib
    zlib/1.2.11
```

Listing 2: *list all versions of a reference*

```
$ conan list zlib
Local Cache
  zlib
    zlib/1.2.11
    zlib/1.2.12
```

As we commented, you can also use the `*` wildcard inside the reference you want to search.

Listing 3: *list all versions of a reference, equivalent to the previous one*

```
$ conan list zlib/*
Local Cache
  zlib
    zlib/1.2.11
    zlib/1.2.12
```

Use the pattern for searching only references matching a specific channel:

Listing 4: *list references with 'stable' channel*

```
$ conan list */*@*/stable
Local Cache
  hello
    hello/2.3.2@mycompany/stable
    hello/1.0.4@mycompany/stable
```

Listing recipe revisions

The shortest way of listing the latest recipe revision for a recipe is using the `name/version@user/channel` as the pattern:

Listing 5: *list latest recipe revision*

```
$ conan list zlib/1.2.11
Local Cache
  zlib
    zlib/1.2.11
      revisions
        ffa77daf83a57094149707928bdce823 (2022-11-02 13:46:53 UTC)
```

This is equivalent to specify explicitly that you want to list the latest recipe revision using the `#latest` placeholder:

Listing 6: *list latest recipe revision*

```
$ conan list zlib/1.2.11#latest
Local Cache
  zlib
    zlib/1.2.11
      revisions
        ffa77daf83a57094149707928bdce823 (2022-11-02 13:46:53 UTC)
```

To list all recipe revisions use the `*` wildcard:

Listing 7: *list all recipe revisions*

```
  $ conan list zlib/1.2.11#*
  Local Cache
    zlib
      zlib/1.2.11
        revisions
          ffa77daf83a57094149707928bdce823 (2022-11-02 13:46:53 UTC)
          8b23adc7acd6f1d6e220338a78e3a19e (2022-10-19 09:19:10 UTC)
          ce3665ce19f82598aa0f7ac0b71ee966 (2022-10-14 11:42:21 UTC)
          31ee767cb2828e539c42913a471e821a (2022-10-12 05:49:39 UTC)
          d77ee68739fcbe5bf37b8a4690eea6ea (2022-08-05 17:17:30 UTC)
```

Listing package IDs

The shortest way of listing all the package IDs belonging to the latest recipe revision is using `name/version@user/channel:*` as the pattern:

Listing 8: *list all package IDs for latest recipe revision*

```
  $ conan list zlib/1.2.11:*
  Local Cache
    zlib
      zlib/1.2.11
        revisions
          d77ee68739fcbe5bf37b8a4690eea6ea (2022-08-05 17:17:30 UTC)
          packages
            d0599452a426a161e02a297c6e0c5070f99b4909
              info
                settings
                  arch: x86_64
                  build_type: Release
                  compiler: apple-clang
                  compiler.version: 12.0
                  os: Macos
                options
                  fPIC: True
                  shared: False
            ebec3dc6d7f6b907b3ada0c3d3cdc83613a2b715
              info
                settings
                  arch: x86_64
                  build_type: Release
```

(continues on next page)

(continued from previous page)

```
                  compiler: gcc
                  compiler.version: 11
                  os: Linux
                options
                  fPIC: True
                  shared: False
```

Note: Here the `#latest` for the recipe revision is implicit, i.e., that pattern is equivalent to `zlib/1.2.11#latest:*`

To list all the package IDs for all the recipe revisions use the `*` wildcard in the revision `#` part:

Listing 9: *list all the package IDs for all the recipe revisions*

```
$ conan list zlib/1.2.11#*:*
zlib
    zlib/1.2.11
      revisions
        d77ee68739fcbe5bf37b8a4690eea6ea (2022-08-05 17:17:30 UTC)
          packages
            d0599452a426a161e02a297c6e0c5070f99b4909
              info
                settings
                  arch: x86_64
                  build_type: Release
                  compiler: apple-clang
                  compiler.version: 12.0
                  os: Macos
                options
                  fPIC: True
                  shared: False
        e4e1703f72ed07c15d73a555ec3a2fa1 (2022-07-04 21:21:45 UTC)
          packages
            d0599452a426a161e02a297c6e0c5070f99b4909
              info
                settings
                  arch: x86_64
                  build_type: Release
                  compiler: apple-clang
                  compiler.version: 12.0
                  os: Macos
                options
                  fPIC: True
                  shared: False
```

Listing package revisions

The shortest way of listing the latest package revision for a specific recipe revision and package ID is using the pattern `name/version@user/channel#rrev:pkgid`

Listing 10: *list latest package revision for a specific recipe revision and package ID*

```
$ conan list zlib/1.2.11
↪#8b23adc7acd6f1d6e220338a78e3a19e:fdb823f07bc228621617c6397210a5c6c4c8807b
Local Cache
  zlib
    zlib/1.2.11
      revisions
        8b23adc7acd6f1d6e220338a78e3a19e (2022-08-05 17:17:30 UTC)
        packages
          fdb823f07bc228621617c6397210a5c6c4c8807b
            revisions
              4834a9b0d050d7cf58c3ab391fe32e25 (2022-11-18 12:33:31 UTC)
```

To list all the package revisions for for the latest recipe revision:

Listing 11: *list all the package revisions for all package-ids the latest recipe revision*

```
$ conan list zlib/1.2.11:*#*
Local Cache
  zlib
    zlib/1.2.11
      revisions
        6a6451bbfcb0e591333827e9784d7dfa (2022-12-29 11:51:39 UTC)
        packages
          b1d267f77ddd5d10d06d2ecf5a6bc433fbb7eeed
            revisions
              67bb089d9d968cbc4ef69e657a03de84 (2022-12-29 11:47:36 UTC)
              5e196dbea832f1efee1e70e058a7eead (2022-12-29 11:47:26 UTC)
              26475a416fa5b61cb962041623748d73 (2022-12-29 11:02:14 UTC)
          d15c4f81b5de757b13ca26b636246edff7bdbf24
            revisions
              a2eb7f4c8f2243b6e80ec9e7ee0e1b25 (2022-12-29 11:51:40 UTC)
```

Note: Here the `#latest` for the recipe revision is implicit, i.e., that pattern is equivalent to `zlib/1.2.11#latest:*#*`

List json output

Note: Best practices

The text output in the terminal should never be parsed or relied on for automation, and it is intended for human reading only. For any automation, the recommended way is using the formatted output as *json*

The `conan list ... --format=json` will return a json output in `stdout` (which can be redirected to a file) with the following structure:

```
$ conan list zlib/1.2.11:*#* --format=json
{
  "Local Cache": {
```

(continues on next page)

(continued from previous page)

```
    "zli/1.0.0": {
      "revisions": {
        "b58eeddfe2fd25ac3a105f72836b3360": {
          "timestamp": "2023-01-10 16:30:27 UTC",
          "packages": {
            "9a4eb3c8701508aa9458b1a73d0633783ecc2270": {
              "revisions": {
                "d9b1e9044ee265092e81db7028ae10e0": {
                  "timestamp": "2023-01-10 22:45:49 UTC"
                }
              },
              "info": {
                "settings": {
                   "os": "Linux"
                }
              }
            },
            "ebec3dc6d7f6b907b3ada0c3d3cdc83613a2b715": {
              "revisions": {
                "d9b1e9044ee265092e81db7028ae10e0": {
                  "timestamp": "2023-01-10 22:45:49 UTC"
                }
              },
              "info": {
                "settings": {
                  "os": "Windows"
                }
              }
            }
          }
        }
      }
    }
  }
}
```

List html output

The `conan list ... --format=html` will return a html output in `stdout` (which can be redirected to a file) with the following structure:

```
$ conan list "zlib/1.2.12#*:*" --format=html
```

Here is the rendered generated HTML.

7.1.7 conan lock

The `conan lock` command contains several subcommands. In addition to these commands, most of the Conan commands that compute a graph, like `create`, `install`, `graph`, can both receive lockfiles as input and produce lockfiles as output.

conan lock add

```
$ conan lock add -h
usage: conan lock add [-h] [-v [V]] [--requires REQUIRES]
                      [--build-requires BUILD_REQUIRES]
                      [--python-requires PYTHON_REQUIRES]
                      [--lockfile-out LOCKFILE_OUT] [--lockfile LOCKFILE]

Add requires, build-requires or python-requires to an existing or new
lockfile. The resulting lockfile will be ordered, newer versions/revisions
first. References can be supplied with and without revisions like "--
requires=pkg/version", but they must be package references, including at least
the version, and they cannot contain a version range.

optional arguments:
  -h, --help            show this help message and exit
  -v [V]                Level of detail of the output. Valid options from less
                        verbose to more verbose: -vquiet, -verror, -vwarning,
                        -vnotice, -vstatus, -v or -vverbose, -vv or -vdebug,
                        -vvv or -vtrace
  --requires REQUIRES   Add references to lockfile.
  --build-requires BUILD_REQUIRES
                        Add build-requires to lockfile
  --python-requires PYTHON_REQUIRES
                        Add python-requires to lockfile
  --lockfile-out LOCKFILE_OUT
                        Filename of the created lockfile
  --lockfile LOCKFILE   Filename of the input lockfile
```

conan lock create

```
$ conan lock create -h
usage: conan lock create [-h] [-v [V]] [--name NAME]
                         [--version VERSION] [--user USER] [--channel CHANNEL]
                         [--requires REQUIRES] [--tool-requires TOOL_REQUIRES]
                         [-b BUILD] [-r REMOTE | -nr] [-u] [-o OPTIONS_HOST]
                         [-o:b OPTIONS_BUILD] [-o:h OPTIONS_HOST]
                         [-pr PROFILE_HOST] [-pr:b PROFILE_BUILD]
                         [-pr:h PROFILE_HOST] [-s SETTINGS_HOST]
                         [-s:b SETTINGS_BUILD] [-s:h SETTINGS_HOST]
                         [-c CONF_HOST] [-c:b CONF_BUILD] [-c:h CONF_HOST]
                         [-l LOCKFILE] [--lockfile-partial]
                         [--lockfile-out LOCKFILE_OUT] [--lockfile-packages]
                         [--lockfile-clean]
                         [path]

Create a lockfile from a conanfile or a reference.

positional arguments:
  path                  Path to a folder containing a recipe (conanfile.py or
                        conanfile.txt) or to a recipe file. e.g.,
                        ./my_project/conanfile.txt.

optional arguments:
  -h, --help            show this help message and exit
  -v [V]                Level of detail of the output. Valid options from less
```

(continues on next page)

(continued from previous page)

```
                        verbose to more verbose: -vquiet, -verror, -vwarning,
                        -vnotice, -vstatus, -v or -vverbose, -vv or -vdebug,
                        -vvv or -vtrace
  --name NAME           Provide a package name if not specified in conanfile
  --version VERSION     Provide a package version if not specified in
                        conanfile
  --user USER           Provide a user if not specified in conanfile
  --channel CHANNEL     Provide a channel if not specified in conanfile
  --requires REQUIRES   Directly provide requires instead of a conanfile
  --tool-requires TOOL_REQUIRES
                        Directly provide tool-requires instead of a conanfile
  -b BUILD, --build BUILD
                        Optional, specify which packages to build from source.
                        Combining multiple '--build' options on one command
                        line is allowed. Possible values: --build="*" Force
                        build from source for all packages. --build=never
                        Disallow build for all packages, use binary packages
                        or fail if a binary package is not found. Cannot be
                        combined with other '--build' options. --build=missing
                        Build packages from source whose binary package is not
                        found. --build=cascade Build packages from source that
                        have at least one dependency being built from source.
                        --build=[pattern] Build packages from source whose
                        package reference matches the pattern. The pattern
                        uses 'fnmatch' style wildcards. --build=![pattern]
                        Excluded packages, which will not be built from the
                        source, whose package reference matches the pattern.
                        The pattern uses 'fnmatch' style wildcards.
                        --build=missing:[pattern] Build from source if a
                        compatible binary does not exist, only for packages
                        matching pattern.
  -r REMOTE, --remote REMOTE
                        Look in the specified remote or remotes server
  -nr, --no-remote      Do not use remote, resolve exclusively in the cache
  -u, --update          Will check the remote and in case a newer version
                        and/or revision of the dependencies exists there, it
                        will install those in the local cache. When using
                        version ranges, it will install the latest version
                        that satisfies the range. Also, if using revisions, it
                        will update to the latest revision for the resolved
                        version range.
  -o OPTIONS_HOST, --options OPTIONS_HOST
                        Define options values (host machine), e.g.: -o
                        Pkg:with_qt=true
  -o:b OPTIONS_BUILD, --options:build OPTIONS_BUILD
                        Define options values (build machine), e.g.: -o:b
                        Pkg:with_qt=true
  -o:h OPTIONS_HOST, --options:host OPTIONS_HOST
                        Define options values (host machine), e.g.: -o:h
                        Pkg:with_qt=true
  -pr PROFILE_HOST, --profile PROFILE_HOST
                        Apply the specified profile to the host machine
  -pr:b PROFILE_BUILD, --profile:build PROFILE_BUILD
                        Apply the specified profile to the build machine
  -pr:h PROFILE_HOST, --profile:host PROFILE_HOST
                        Apply the specified profile to the host machine
  -s SETTINGS_HOST, --settings SETTINGS_HOST
```

(continues on next page)

(continued from previous page)

```
                        Settings to build the package, overwriting the
                        defaults (host machine). e.g.: -s compiler=gcc
  -s:b SETTINGS_BUILD, --settings:build SETTINGS_BUILD
                        Settings to build the package, overwriting the
                        defaults (build machine). e.g.: -s:b compiler=gcc
  -s:h SETTINGS_HOST, --settings:host SETTINGS_HOST
                        Settings to build the package, overwriting the
                        defaults (host machine). e.g.: -s:h compiler=gcc
  -c CONF_HOST, --conf CONF_HOST
                        Configuration to build the package, overwriting the
                        defaults (host machine). e.g.: -c
                        tools.cmake.cmaketoolchain:generator=Xcode
  -c:b CONF_BUILD, --conf:build CONF_BUILD
                        Configuration to build the package, overwriting the
                        defaults (build machine). e.g.: -c:b
                        tools.cmake.cmaketoolchain:generator=Xcode
  -c:h CONF_HOST, --conf:host CONF_HOST
                        Configuration to build the package, overwriting the
                        defaults (host machine). e.g.: -c:h
                        tools.cmake.cmaketoolchain:generator=Xcode
  -l LOCKFILE, --lockfile LOCKFILE
                        Path to a lockfile. Use --lockfile="" to avoid
                        automatic use of existing 'conan.lock' file
  --lockfile-partial    Do not raise an error if some dependency is not found
                        in lockfile
  --lockfile-out LOCKFILE_OUT
                        Filename of the updated lockfile
  --lockfile-packages   Lock package-id and package-revision information
  --lockfile-clean      Remove unused entries from the lockfile
```

The `conan lock create` command creates a lockfile for the recipe or reference specified in `path`.

conan lock merge

```
$ conan lock merge -h
usage: conan lock merge [-h] [-v [V]] [--lockfile LOCKFILE]
                        [--lockfile-out LOCKFILE_OUT]

Merge 2 or more lockfiles.

optional arguments:
  -h, --help            show this help message and exit
  -v [V]                Level of detail of the output. Valid options from less
                        verbose to more verbose: -vquiet, -verror, -vwarning,
                        -vnotice, -vstatus, -v or -vverbose, -vv or -vdebug,
                        -vvv or -vtrace
  --lockfile LOCKFILE   Path to lockfile to be merged
  --lockfile-out LOCKFILE_OUT
                        Filename of the created lockfile
```

- *conan lock add*: Manually add items to a lockfile
- *conan lock create*: Evaluates a dependency graph and save a lockfile
- *conan lock merge*: Merge several existing lockfiles into one.

7.1.8 conan profile

Manage profiles

conan profile detect

```
$ conan profile detect -h
usage: conan profile detect [-h] [-v [V]] [--name NAME] [-f]

Generate a profile using auto-detected values.

optional arguments:
  -h, --help   show this help message and exit
  -v [V]       Level of detail of the output. Valid options from less verbose
               to more verbose: -vquiet, -verror, -vwarning, -vnotice,
               -vstatus, -v or -vverbose, -vv or -vdebug, -vvv or -vtrace
  --name NAME  Profile name, 'default' if not specified
  -f, --force  Overwrite if exists
```

Warning: The output of `conan profile detect` is **not stable**. It can change at any time in future Conan releases to adapt to latest tools, latest versions, or other changes in the environment. See *the Conan stability* section for more information.

Note: **Best practices** It is not recommended to use `conan profile detect` in production. To guarantee reproducibility, it is recommended to define your own profiles, store them in a git repo or in a zip in a server, and distribute it to your team and CI machines with `conan config install`, together with other configuration like custom settings, custom remotes definition, etc.

conan profile list

```
$ conan profile list -h
usage: conan profile list [-h] [-f FORMAT] [-v [V]]

List all profiles in the cache.

optional arguments:
  -h, --help            show this help message and exit
  -f FORMAT, --format FORMAT
                        Select the output format: json
  -v [V]                Level of detail of the output. Valid options from less
                        verbose to more verbose: -vquiet, -verror, -vwarning,
                        -vnotice, -vstatus, -v or -vverbose, -vv or -vdebug,
                        -vvv or -vtrace
```

conan profile path

```
$ conan profile path -h
usage: conan profile path [-h] [-v [V]] [-o OPTIONS_HOST]
```

(continues on next page)

(continued from previous page)

```
                        [-o:b OPTIONS_BUILD] [-o:h OPTIONS_HOST]
                        [-pr PROFILE_HOST] [-pr:b PROFILE_BUILD]
                        [-pr:h PROFILE_HOST] [-s SETTINGS_HOST]
                        [-s:b SETTINGS_BUILD] [-s:h SETTINGS_HOST]
                        [-c CONF_HOST] [-c:b CONF_BUILD] [-c:h CONF_HOST]
                        name

Show profile path location.

positional arguments:
  name                  Profile name

optional arguments:
  -h, --help            show this help message and exit
  -v [V]                Level of detail of the output. Valid options from less
                        verbose to more verbose: -vquiet, -verror, -vwarning,
                        -vnotice, -vstatus, -v or -vverbose, -vv or -vdebug,
                        -vvv or -vtrace
  -o OPTIONS_HOST, --options OPTIONS_HOST
                        Define options values (host machine), e.g.: -o
                        Pkg:with_qt=true
  -o:b OPTIONS_BUILD, --options:build OPTIONS_BUILD
                        Define options values (build machine), e.g.: -o:b
                        Pkg:with_qt=true
  -o:h OPTIONS_HOST, --options:host OPTIONS_HOST
                        Define options values (host machine), e.g.: -o:h
                        Pkg:with_qt=true
  -pr PROFILE_HOST, --profile PROFILE_HOST
                        Apply the specified profile to the host machine
  -pr:b PROFILE_BUILD, --profile:build PROFILE_BUILD
                        Apply the specified profile to the build machine
  -pr:h PROFILE_HOST, --profile:host PROFILE_HOST
                        Apply the specified profile to the host machine
  -s SETTINGS_HOST, --settings SETTINGS_HOST
                        Settings to build the package, overwriting the
                        defaults (host machine). e.g.: -s compiler=gcc
  -s:b SETTINGS_BUILD, --settings:build SETTINGS_BUILD
                        Settings to build the package, overwriting the
                        defaults (build machine). e.g.: -s:b compiler=gcc
  -s:h SETTINGS_HOST, --settings:host SETTINGS_HOST
                        Settings to build the package, overwriting the
                        defaults (host machine). e.g.: -s:h compiler=gcc
  -c CONF_HOST, --conf CONF_HOST
                        Configuration to build the package, overwriting the
                        defaults (host machine). e.g.: -c
                        tools.cmake.cmaketoolchain:generator=Xcode
  -c:b CONF_BUILD, --conf:build CONF_BUILD
                        Configuration to build the package, overwriting the
                        defaults (build machine). e.g.: -c:b
                        tools.cmake.cmaketoolchain:generator=Xcode
  -c:h CONF_HOST, --conf:host CONF_HOST
                        Configuration to build the package, overwriting the
                        defaults (host machine). e.g.: -c:h
                        tools.cmake.cmaketoolchain:generator=Xcod
```

conan profile show

```
$ conan profile show -h
usage: conan profile show [-h] [-v [V]] [-o OPTIONS_HOST]
                          [-o:b OPTIONS_BUILD] [-o:h OPTIONS_HOST]
                          [-pr PROFILE_HOST] [-pr:b PROFILE_BUILD]
                          [-pr:h PROFILE_HOST] [-s SETTINGS_HOST]
                          [-s:b SETTINGS_BUILD] [-s:h SETTINGS_HOST]
                          [-c CONF_HOST] [-c:b CONF_BUILD] [-c:h CONF_HOST]

Show aggregated profiles from the passed arguments.

optional arguments:
  -h, --help            show this help message and exit
  -v [V]                Level of detail of the output. Valid options from less
                        verbose to more verbose: -vquiet, -verror, -vwarning,
                        -vnotice, -vstatus, -v or -vverbose, -vv or -vdebug,
                        -vvv or -vtrace
  -o OPTIONS_HOST, --options OPTIONS_HOST
                        Define options values (host machine), e.g.: -o
                        Pkg:with_qt=true
  -o:b OPTIONS_BUILD, --options:build OPTIONS_BUILD
                        Define options values (build machine), e.g.: -o:b
                        Pkg:with_qt=true
  -o:h OPTIONS_HOST, --options:host OPTIONS_HOST
                        Define options values (host machine), e.g.: -o:h
                        Pkg:with_qt=true
  -pr PROFILE_HOST, --profile PROFILE_HOST
                        Apply the specified profile to the host machine
  -pr:b PROFILE_BUILD, --profile:build PROFILE_BUILD
                        Apply the specified profile to the build machine
  -pr:h PROFILE_HOST, --profile:host PROFILE_HOST
                        Apply the specified profile to the host machine
  -s SETTINGS_HOST, --settings SETTINGS_HOST
                        Settings to build the package, overwriting the
                        defaults (host machine). e.g.: -s compiler=gcc
  -s:b SETTINGS_BUILD, --settings:build SETTINGS_BUILD
                        Settings to build the package, overwriting the
                        defaults (build machine). e.g.: -s:b compiler=gcc
  -s:h SETTINGS_HOST, --settings:host SETTINGS_HOST
                        Settings to build the package, overwriting the
                        defaults (host machine). e.g.: -s:h compiler=gcc
  -c CONF_HOST, --conf CONF_HOST
                        Configuration to build the package, overwriting the
                        defaults (host machine). e.g.: -c
                        tools.cmake.cmaketoolchain:generator=Xcode
  -c:b CONF_BUILD, --conf:build CONF_BUILD
                        Configuration to build the package, overwriting the
                        defaults (build machine). e.g.: -c:b
                        tools.cmake.cmaketoolchain:generator=Xcode
  -c:h CONF_HOST, --conf:host CONF_HOST
                        Configuration to build the package, overwriting the
                        defaults (host machine). e.g.: -c:h
                        tools.cmake.cmaketoolchain:generator=Xcode
```

7.1.9 conan remove

```
$ conan remove -h
usage: conan remove [-h] [-v [V]] [-c] [-p PACKAGE_QUERY]
                    [-r REMOTE]
                    reference

Remove recipes or packages from local cache or a remote.

- If no remote is specified (-r), the removal will be done in the local conan cache.
- If a recipe reference is specified, it will remove the recipe and all the packages,
↪unless -p
  is specified, in that case, only the packages matching the specified query (and not
↪the recipe)
  will be removed.
- If a package reference is specified, it will remove only the package.

positional arguments:
  reference             Recipe reference or package reference, can contain *
                        aswildcard at any reference field. e.g: lib/*

optional arguments:
  -h, --help            show this help message and exit
  -v [V]                Level of detail of the output. Valid options from less
                        verbose to more verbose: -vquiet, -verror, -vwarning,
                        -vnotice, -vstatus, -v or -vverbose, -vv or -vdebug,
                        -vvv or -vtrace
  -c, --confirm         Remove without requesting a confirmation
  -p PACKAGE_QUERY, --package-query PACKAGE_QUERY
                        Remove all packages (empty) or provide a query:
                        os=Windows AND (arch=x86 OR compiler=gcc)
  -r REMOTE, --remote REMOTE
                        Will remove from the specified remote
```

The `conan remove` command removes recipes and packages from the local cache or from a specified remote. Depending on the patterns specified as argument, it is possible to remove a complete package, or just remove the binaries, leaving still the recipe available.

To remove recipes and their associated package binaries from the local cache:

```
$ conan remove "*"
# Removes everything from the cache

$ conan remove zlib/*
# Remove all possible versions of zlib, including all recipes, revisions and packages

$ conan remove zlib/1.2.11
# Remove zlib/1.2.11, all its revisions and package binaries. Leave other zlib
↪versions

$ conan remove zlib/1.2.11#latest
# Remove zlib/1.2.11, only its latest recipe revision and binaries of that revision
# Leave the other zlib/1.2.11 revisions intact

$ conan remove zlib/1.2.11#<revision>
# Remove zlib/1.2.11, only its exact <revision> and binaries of that revision
# Leave the other zlib/1.2.11 revisions intact
```

To remove only package binaries, but leaving the recipes, it is necessary to specify the pattern including the `:` separator of the `package_id`:

```
$ conan remove zlib/1.2.11:*
# Removes all the zlib/1.2.11 package binaries from all the recipe revisions

$ conan remove zlib/*:*
# Removes all the binaries from all the recipe revisions from all zlib versions

$ conan remove zlib/1.2.11#latest:*
# Removes all the zlib/1.2.11 package binaries only from the latest zlib/1.2.11␣
↪recipe revision

$ conan remove zlib/1.2.11:<package_id>
# Removes the package binary <package_id> from all the zlib/1.2.11 recipe revisions

$ conan remove zlib/1.2.11:#latest<package_id>#latest
# Removes only the latest package revision of the binary identified with <package_id>
# from the latest recipe revision of zlib/1.2.11
# WARNING: Recall that having more than 1 package revision is a smell and shouldn't␣
↪happen
# in normal situations
```

All the above commands, by default operate in the Conan cache. To remove artifacts from a server, use the `-r=myremote` argument:

```
$ conan remove zlib/1.2.11:* -r=myremote
# Removes all the zlib/1.2.11 package binaries from all the recipe revisions in
# the remote <myremote>
```

7.1.10 conan remote

Use this command to add, edit and remove Conan repositories from the Conan remote registry and also manage authentication to those remotes. For more information on how to work with Conan repositories, please check the *dedicated section*.

```
$ conan remote -h
usage: conan remote [-h] [-v [V]] {add,auth,disable,enable,list,list-users,login,
↪logout,remove,rename,set-user,update} ...

Manage the remote list and the users authenticated on them.

positional arguments:
  {add,auth,disable,enable,list,list-users,login,logout,remove,rename,set-user,update}
                        sub-command help
    add                 Add a remote.
    auth                Authenticate in the defined remotes
    disable             Disable all the remotes matching a pattern.
    enable              Enable all the remotes matching a pattern.
    list                List current remotes.
    list-users          List the users logged into all the remotes.
    login               Login into the specified remotes matching a pattern.
    logout              Clear the existing credentials for the specified remotes␣
↪matching a pattern.
    remove              Remove a remote.
    rename              Rename a remote.
```

(continues on next page)

(continued from previous page)

```
    set-user            Associate a username with a remote matching a pattern without␣
↪performing the authentication.
    update              Update a remote.

options:
  -h, --help            show this help message and exit
  -v [V]                Level of detail of the output. Valid options from less␣
↪verbose to more verbose: -vquiet, -verror, -vwarning,
                        -vnotice, -vstatus, -v or -vverbose, -vv or -vdebug, -vvv or -
↪vtrace
```

conan remote add

```
$ conan remote add -h
usage: conan remote add [-h] [-v [V]] [--insecure] [--index INDEX] [-f] name url

Add a remote.

positional arguments:
  name             Name of the remote to add
  url              Url of the remote

options:
  -h, --help       show this help message and exit
  -v [V]           Level of detail of the output. Valid options from less verbose to␣
↪more verbose:
                   -vquiet, -verror, -vwarning, -vnotice, -vstatus, -v or -vverbose, -
↪vv or -vdebug, -vvv
                   or -vtrace
  --insecure       Allow insecure server connections when using SSL
  --index INDEX    Insert the remote at a specific position in the remote list
  -f, --force      Force the definition of the remote even if duplicated
```

conan remote auth

```
$ conan remote auth -h
usage: conan remote auth [-h] [-v [V]] [--with-user] remote

Authenticate in the defined remotes

positional arguments:
  remote         Pattern or name of the remote/s to authenticate against. The
                 pattern uses 'fnmatch' style wildcards.

optional arguments:
  -h, --help     show this help message and exit
  -v [V]         Level of detail of the output. Valid options from less verbose
                 to more verbose: -vquiet, -verror, -vwarning, -vnotice,
                 -vstatus, -v or -vverbose, -vv or -vdebug, -vvv or -vtrace
  --with-user    Only try to auth in those remotes that already have a username
                 or a CONAN_LOGIN_ env-var defined
```

conan remote disable

```
$ conan remote disable -h
usage: conan remote disable [-h] [-v [V]] remote

Disable all the remotes matching a pattern.

positional arguments:
  remote      Pattern of the remote/s to disable. The pattern uses 'fnmatch'
              style wildcards.

options:
  -h, --help  show this help message and exit
  -v [V]      Level of detail of the output. Valid options from less verbose
              to more verbose: -vquiet, -verror, -vwarning, -vnotice,
              -vstatus, -v or -vverbose, -vv or -vdebug, -vvv or -vtrace
```

conan remote enable

```
$ conan remote enable -h
usage: conan remote enable [-h] [-v [V]] remote

Enable all the remotes matching a pattern.

positional arguments:
  remote      Pattern of the remote/s to enable. The pattern uses 'fnmatch'
              style wildcards.

options:
  -h, --help  show this help message and exit
  -v [V]      Level of detail of the output. Valid options from less verbose
              to more verbose: -vquiet, -verror, -vwarning, -vnotice,
              -vstatus, -v or -vverbose, -vv or -vdebug, -vvv or -vtrace
```

conan remote list

```
$ conan remote list -h
usage: conan remote list [-h] [-f FORMAT] [-v [V]]

List current remotes.

options:
  -h, --help            show this help message and exit
  -f FORMAT, --format FORMAT
                        Select the output format: json
  -v [V]                Level of detail of the output. Valid options from less
                        verbose to more verbose: -vquiet, -verror, -vwarning,
                        -vnotice, -vstatus, -v or -vverbose, -vv or -vdebug,
                        -vvv or -vtrace
```

conan remote list-users

```
$ conan remote list-users -h
usage: conan remote list-users [-h] [-f FORMAT] [-v [V]]

List the users logged into all the remotes.

options:
  -h, --help            show this help message and exit
  -f FORMAT, --format FORMAT
                        Select the output format: json
  -v [V]                Level of detail of the output. Valid options from less
                        verbose to more verbose: -vquiet, -verror, -vwarning,
                        -vnotice, -vstatus, -v or -vverbose, -vv or -vdebug,
                        -vvv or -vtrace
```

conan remote login

```
$ conan remote login -h
usage: conan remote login [-h] [-f FORMAT] [-v [V]] [-p [PASSWORD]]
                          remote username

Login into the specified remotes matching a pattern.

positional arguments:
  remote                Pattern or name of the remote to login into. The
                        pattern uses 'fnmatch' style wildcards.
  username              Username

options:
  -h, --help            show this help message and exit
  -f FORMAT, --format FORMAT
                        Select the output format: json
  -v [V]                Level of detail of the output. Valid options from less
                        verbose to more verbose: -vquiet, -verror, -vwarning,
                        -vnotice, -vstatus, -v or -vverbose, -vv or -vdebug,
                        -vvv or -vtrace
  -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)
```

conan remote logout

```
$ conan remote logout -h
usage: conan remote logout [-h] [-f FORMAT] [-v [V]] remote

Clear the existing credentials for the specified remotes matching a pattern.

positional arguments:
  remote                Pattern or name of the remote to logout. The pattern
                        uses 'fnmatch' style wildcards.

options:
```

(continues on next page)

(continued from previous page)

```
  -h, --help            show this help message and exit
  -f FORMAT, --format FORMAT
                        Select the output format: json
  -v [V]                Level of detail of the output. Valid options from less
                        verbose to more verbose: -vquiet, -verror, -vwarning,
                        -vnotice, -vstatus, -v or -vverbose, -vv or -vdebug,
                        -vvv or -vtrace
```

conan remote remove

```
$ conan remote remove -h
usage: conan remote remove [-h] [-v [V]] remote

Remove a remote.

positional arguments:
  remote      Name of the remote to remove. Accepts 'fnmatch' style wildcards.

options:
  -h, --help  show this help message and exit
  -v [V]      Level of detail of the output. Valid options from less verbose
              to more verbose: -vquiet, -verror, -vwarning, -vnotice,
              -vstatus, -v or -vverbose, -vv or -vdebug, -vvv or -vtrace
```

conan remote rename

```
$ conan remote rename -h
usage: conan remote rename [-h] [-v [V]] remote new_name

Rename a remote.

positional arguments:
  remote      Current name of the remote
  new_name    New name for the remote

options:
  -h, --help  show this help message and exit
  -v [V]      Level of detail of the output. Valid options from less verbose
              to more verbose: -vquiet, -verror, -vwarning, -vnotice,
              -vstatus, -v or -vverbose, -vv or -vdebug, -vvv or -vtrace
```

conan remote set-user

```
$ conan remote set-user -h
usage: conan remote set-user [-h] [-f FORMAT] [-v [V]]
                             remote username

Associate a username with a remote matching a pattern without performing the
authentication.

positional arguments:
```

(continues on next page)

(continued from previous page)

```
  remote                Pattern or name of the remote. The pattern uses
                        'fnmatch' style wildcards.
  username              Username

options:
  -h, --help            show this help message and exit
  -f FORMAT, --format FORMAT
                        Select the output format: json
  -v [V]                Level of detail of the output. Valid options from less
                        verbose to more verbose: -vquiet, -verror, -vwarning,
                        -vnotice, -vstatus, -v or -vverbose, -vv or -vdebug,
                        -vvv or -vtrace
```

conan remote update

```
$ conan remote update -h
usage: conan remote update [-h] [-v [V]] [--url URL] [--secure]
                           [--insecure] [--index INDEX]
                           remote

Update a remote.

positional arguments:
  remote          Name of the remote to update

options:
  -h, --help      show this help message and exit
  -v [V]          Level of detail of the output. Valid options from less
                  verbose to more verbose: -vquiet, -verror, -vwarning,
                  -vnotice, -vstatus, -v or -vverbose, -vv or -vdebug, -vvv or
                  -vtrace
  --url URL       New url for the remote
  --secure        Don't allow insecure server connections when using SSL
  --insecure      Allow insecure server connections when using SSL
  --index INDEX   Insert the remote at a specific position in the remote list
```

Read more

- *Uploading packages tutorial*
- *Working with Conan repositories*
- *Upload Conan packages to remotes using conan upload command*

7.1.11 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 FORMAT] [-v [V]] [-r REMOTE] reference
```

(continues on next page)

(continued from previous page)

```
Search for package recipes in all the remotes (by default), or a remote.

positional arguments:
  reference             Recipe reference to search for.It can contain * as
                        wildcard at any reference field.

optional arguments:
  -h, --help            show this help message and exit
  -f FORMAT, --format FORMAT
                        Select the output format: json
  -v [V]                Level of detail of the output. Valid options from less
                        verbose to more verbose: -vquiet, -verror, -vwarning,
                        -vnotice, -vstatus, -v or -vverbose, -vv or -vdebug,
                        -vvv or -vtrace
  -r REMOTE, --remote REMOTE
                        Remote names. Accepts wildcards. If not specified it
                        searches in all the 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
[
    {
        "remote": "conancenter",
        "error": null,
        "results": [
            {
                "name": "zlib",
                "id": "zlib/1.2.11"
            }
        ]
    }
]
```

- *conan cache*: Return the path of recipes and packages in the cache
- *conan config*: Manage Conan configuration (remotes, settings, plugins, etc)
- *conan graph*: Obtain information about the dependency graph without fetching binaries
- *conan inspect*: Inspect a conanfile.py to return the public fields
- *conan install*: Install dependencies

- *conan list*: List recipes, revisions and packages in the local cache or in remotes
- *conan lock*: Create and manage lockfiles
- *conan profile*: Display and manage profile files
- *conan remove*: Remove packages from the local cache or from remotes
- *conan remote*: Add, remove, login/logout and manage remote server
- *conan search*: Search packages matching a name

Creator commands:

7.1.12 conan build

```
$ conan build -h
usage: conan build [-h] [-v [V]] [--name NAME] [--version VERSION]
                   [--user USER] [--channel CHANNEL] [-of OUTPUT_FOLDER]
                   [-b BUILD] [-r REMOTE | -nr] [-u] [-o OPTIONS_HOST]
                   [-o:b OPTIONS_BUILD] [-o:h OPTIONS_HOST] [-pr PROFILE_HOST]
                   [-pr:b PROFILE_BUILD] [-pr:h PROFILE_HOST]
                   [-s SETTINGS_HOST] [-s:b SETTINGS_BUILD]
                   [-s:h SETTINGS_HOST] [-c CONF_HOST] [-c:b CONF_BUILD]
                   [-c:h CONF_HOST] [-l LOCKFILE] [--lockfile-partial]
                   [--lockfile-out LOCKFILE_OUT] [--lockfile-packages]
                   [--lockfile-clean]
                   [path]

Install dependencies and call the build() method.

positional arguments:
  path                  Path to a python-based recipe file or a folder
                        containing a conanfile.py recipe. conanfile.txt cannot
                        be used with conan build.

optional arguments:
  -h, --help            show this help message and exit
  -v [V]                Level of detail of the output. Valid options from less
                        verbose to more verbose: -vquiet, -verror, -vwarning,
                        -vnotice, -vstatus, -v or -vverbose, -vv or -vdebug,
                        -vvv or -vtrace
  --name NAME           Provide a package name if not specified in conanfile
  --version VERSION     Provide a package version if not specified in
                        conanfile
  --user USER           Provide a user if not specified in conanfile
  --channel CHANNEL     Provide a channel if not specified in conanfile
  -of OUTPUT_FOLDER, --output-folder OUTPUT_FOLDER
                        The root output folder for generated and build files
  -b BUILD, --build BUILD
                        Optional, specify which packages to build from source.
                        Combining multiple '--build' options on one command
                        line is allowed. Possible values: --build="*" Force
                        build from source for all packages. --build=never
                        Disallow build for all packages, use binary packages
                        or fail if a binary package is not found. Cannot be
                        combined with other '--build' options. --build=missing
                        Build packages from source whose binary package is not
                        found. --build=cascade Build packages from source that
```

(continues on next page)

(continued from previous page)

```
                        have at least one dependency being built from source.
                        --build=[pattern] Build packages from source whose
                        package reference matches the pattern. The pattern
                        uses 'fnmatch' style wildcards. --build=![pattern]
                        Excluded packages, which will not be built from the
                        source, whose package reference matches the pattern.
                        The pattern uses 'fnmatch' style wildcards.
                        --build=missing:[pattern] Build from source if a
                        compatible binary does not exist, only for packages
                        matching pattern.
  -r REMOTE, --remote REMOTE
                        Look in the specified remote or remotes server
  -nr, --no-remote      Do not use remote, resolve exclusively in the cache
  -u, --update          Will check the remote and in case a newer version
                        and/or revision of the dependencies exists there, it
                        will install those in the local cache. When using
                        version ranges, it will install the latest version
                        that satisfies the range. Also, if using revisions, it
                        will update to the latest revision for the resolved
                        version range.
  -o OPTIONS_HOST, --options OPTIONS_HOST
                        Define options values (host machine), e.g.: -o
                        Pkg:with_qt=true
  -o:b OPTIONS_BUILD, --options:build OPTIONS_BUILD
                        Define options values (build machine), e.g.: -o:b
                        Pkg:with_qt=true
  -o:h OPTIONS_HOST, --options:host OPTIONS_HOST
                        Define options values (host machine), e.g.: -o:h
                        Pkg:with_qt=true
  -pr PROFILE_HOST, --profile PROFILE_HOST
                        Apply the specified profile to the host machine
  -pr:b PROFILE_BUILD, --profile:build PROFILE_BUILD
                        Apply the specified profile to the build machine
  -pr:h PROFILE_HOST, --profile:host PROFILE_HOST
                        Apply the specified profile to the host machine
  -s SETTINGS_HOST, --settings SETTINGS_HOST
                        Settings to build the package, overwriting the
                        defaults (host machine). e.g.: -s compiler=gcc
  -s:b SETTINGS_BUILD, --settings:build SETTINGS_BUILD
                        Settings to build the package, overwriting the
                        defaults (build machine). e.g.: -s:b compiler=gcc
  -s:h SETTINGS_HOST, --settings:host SETTINGS_HOST
                        Settings to build the package, overwriting the
                        defaults (host machine). e.g.: -s:h compiler=gcc
  -c CONF_HOST, --conf CONF_HOST
                        Configuration to build the package, overwriting the
                        defaults (host machine). e.g.: -c
                        tools.cmake.cmaketoolchain:generator=Xcode
  -c:b CONF_BUILD, --conf:build CONF_BUILD
                        Configuration to build the package, overwriting the
                        defaults (build machine). e.g.: -c:b
                        tools.cmake.cmaketoolchain:generator=Xcode
  -c:h CONF_HOST, --conf:host CONF_HOST
                        Configuration to build the package, overwriting the
                        defaults (host machine). e.g.: -c:h
                        tools.cmake.cmaketoolchain:generator=Xcode
  -l LOCKFILE, --lockfile LOCKFILE
```

(continues on next page)

(continued from previous page)

```
                        Path to a lockfile. Use --lockfile="" to avoid
                        automatic use of existing 'conan.lock' file
  --lockfile-partial    Do not raise an error if some dependency is not found
                        in lockfile
  --lockfile-out LOCKFILE_OUT
                        Filename of the updated lockfile
  --lockfile-packages   Lock package-id and package-revision information
  --lockfile-clean      Remove unused entries from the lockfile
```

The `conan build` command installs the recipe specified in `path` and calls its `build` method.

See also:

- Read the tutorial about the *local package developement flow*.

7.1.13 conan create

```
$ conan create -h
usage: conan create [-h] [-f FORMAT] [-v [V]] [--name NAME]
                    [--version VERSION] [--user USER] [--channel CHANNEL]
                    [-l LOCKFILE] [--lockfile-partial]
                    [--lockfile-out LOCKFILE_OUT] [--lockfile-packages]
                    [--lockfile-clean] [-b BUILD] [-r REMOTE | -nr] [-u]
                    [-o OPTIONS_HOST] [-o:b OPTIONS_BUILD] [-o:h OPTIONS_HOST]
                    [-pr PROFILE_HOST] [-pr:b PROFILE_BUILD]
                    [-pr:h PROFILE_HOST] [-s SETTINGS_HOST]
                    [-s:b SETTINGS_BUILD] [-s:h SETTINGS_HOST] [-c CONF_HOST]
                    [-c:b CONF_BUILD] [-c:h CONF_HOST] [--build-require]
                    [-tf TEST_FOLDER]
                    path

Create a package.

positional arguments:
  path                  Path to a folder containing a recipe (conanfile.py)

optional arguments:
  -h, --help            show this help message and exit
  -f FORMAT, --format FORMAT
                        Select the output format: json
  -v [V]                Level of detail of the output. Valid options from less
                        verbose to more verbose: -vquiet, -verror, -vwarning,
                        -vnotice, -vstatus, -v or -vverbose, -vv or -vdebug,
                        -vvv or -vtrace
  --name NAME           Provide a package name if not specified in conanfile
  --version VERSION     Provide a package version if not specified in
                        conanfile
  --user USER           Provide a user if not specified in conanfile
  --channel CHANNEL     Provide a channel if not specified in conanfile
  -l LOCKFILE, --lockfile LOCKFILE
                        Path to a lockfile. Use --lockfile="" to avoid
                        automatic use of existing 'conan.lock' file
  --lockfile-partial    Do not raise an error if some dependency is not found
                        in lockfile
  --lockfile-out LOCKFILE_OUT
                        Filename of the updated lockfile
```

(continues on next page)

(continued from previous page)

```
  --lockfile-packages   Lock package-id and package-revision information
  --lockfile-clean      Remove unused entries from the lockfile
  -b BUILD, --build BUILD
                        Optional, specify which packages to build from source.
                        Combining multiple '--build' options on one command
                        line is allowed. Possible values: --build="*" Force
                        build from source for all packages. --build=never
                        Disallow build for all packages, use binary packages
                        or fail if a binary package is not found. Cannot be
                        combined with other '--build' options. --build=missing
                        Build packages from source whose binary package is not
                        found. --build=cascade Build packages from source that
                        have at least one dependency being built from source.
                        --build=[pattern] Build packages from source whose
                        package reference matches the pattern. The pattern
                        uses 'fnmatch' style wildcards. --build=![pattern]
                        Excluded packages, which will not be built from the
                        source, whose package reference matches the pattern.
                        The pattern uses 'fnmatch' style wildcards.
                        --build=missing:[pattern] Build from source if a
                        compatible binary does not exist, only for packages
                        matching pattern.
  -r REMOTE, --remote REMOTE
                        Look in the specified remote or remotes server
  -nr, --no-remote      Do not use remote, resolve exclusively in the cache
  -u, --update          Will check the remote and in case a newer version
                        and/or revision of the dependencies exists there, it
                        will install those in the local cache. When using
                        version ranges, it will install the latest version
                        that satisfies the range. Also, if using revisions, it
                        will update to the latest revision for the resolved
                        version range.
  -o OPTIONS_HOST, --options OPTIONS_HOST
                        Define options values (host machine), e.g.: -o
                        Pkg:with_qt=true
  -o:b OPTIONS_BUILD, --options:build OPTIONS_BUILD
                        Define options values (build machine), e.g.: -o:b
                        Pkg:with_qt=true
  -o:h OPTIONS_HOST, --options:host OPTIONS_HOST
                        Define options values (host machine), e.g.: -o:h
                        Pkg:with_qt=true
  -pr PROFILE_HOST, --profile PROFILE_HOST
                        Apply the specified profile to the host machine
  -pr:b PROFILE_BUILD, --profile:build PROFILE_BUILD
                        Apply the specified profile to the build machine
  -pr:h PROFILE_HOST, --profile:host PROFILE_HOST
                        Apply the specified profile to the host machine
  -s SETTINGS_HOST, --settings SETTINGS_HOST
                        Settings to build the package, overwriting the
                        defaults (host machine). e.g.: -s compiler=gcc
  -s:b SETTINGS_BUILD, --settings:build SETTINGS_BUILD
                        Settings to build the package, overwriting the
                        defaults (build machine). e.g.: -s:b compiler=gcc
  -s:h SETTINGS_HOST, --settings:host SETTINGS_HOST
                        Settings to build the package, overwriting the
                        defaults (host machine). e.g.: -s:h compiler=gcc
  -c CONF_HOST, --conf CONF_HOST
```

(continues on next page)

(continued from previous page)

```
                        Configuration to build the package, overwriting the
                        defaults (host machine). e.g.: -c
                        tools.cmake.cmaketoolchain:generator=Xcode
  -c:b CONF_BUILD, --conf:build CONF_BUILD
                        Configuration to build the package, overwriting the
                        defaults (build machine). e.g.: -c:b
                        tools.cmake.cmaketoolchain:generator=Xcode
  -c:h CONF_HOST, --conf:host CONF_HOST
                        Configuration to build the package, overwriting the
                        defaults (host machine). e.g.: -c:h
                        tools.cmake.cmaketoolchain:generator=Xcode
  --build-require       Whether the provided reference is a build-require
  -tf TEST_FOLDER, --test-folder TEST_FOLDER
                        Alternative test folder name. By default it is
                        "test_package". Use "" to skip the test stage
```

The `conan create` command creates a package from the recipe specified in `path`.

Warning: The json output of the `conan create --format=json` is **experimental** and subject to change.

7.1.14 conan download

```
$ conan download -h
usage: conan download [-h] [-v [V]] [--only-recipe]
                      [-p PACKAGE_QUERY] -r REMOTE
                      reference

Download (without installing) a single conan package from a remote server.

It downloads just the package, but not its transitive dependencies, and it will not␣
↪call
any generate, generators or deployers.
It can download multiple packages if patterns are used, and also works with queries␣
↪over
the package binaries.

positional arguments:
  reference             Recipe reference or package reference, can contain *
                        as wildcard at any reference field. If revision is not
                        specified, it is assumed latest one.

optional arguments:
  -h, --help            show this help message and exit
  -v [V]                Level of detail of the output. Valid options from less
                        verbose to more verbose: -vquiet, -verror, -vwarning,
                        -vnotice, -vstatus, -v or -vverbose, -vv or -vdebug,
                        -vvv or -vtrace
  --only-recipe         Download only the recipe/s, not the binary packages.
  -p PACKAGE_QUERY, --package-query PACKAGE_QUERY
                        Only upload packages matching a specific query. e.g:
                        os=Windows AND (arch=x86 OR compiler=gcc)
  -r REMOTE, --remote REMOTE
                        Download from this specific remote
```

7.1.15 conan editable

Allow working with a package that resides in user folder.

conan editable add

```
$ conan editable add -h
usage: conan editable add [-h] [-v [V]] [--name NAME]
                          [--version VERSION] [--user USER]
                          [--channel CHANNEL] [-of OUTPUT_FOLDER]
                          path

Define the given <path> location as the package <reference>, so when this
package is required, it is used from this <path> location instead of the
cache.

positional arguments:
  path                  Path to the package folder in the user workspace

optional arguments:
  -h, --help            show this help message and exit
  -v [V]                Level of detail of the output. Valid options from less
                        verbose to more verbose: -vquiet, -verror, -vwarning,
                        -vnotice, -vstatus, -v or -vverbose, -vv or -vdebug,
                        -vvv or -vtrace
  --name NAME           Provide a package name if not specified in conanfile
  --version VERSION     Provide a package version if not specified in
                        conanfile
  --user USER           Provide a user if not specified in conanfile
  --channel CHANNEL     Provide a channel if not specified in conanfile
  -of OUTPUT_FOLDER, --output-folder OUTPUT_FOLDER
                        The root output folder for generated and build files
```

conan editable remove

```
$ conan editable remove -h
usage: conan editable remove [-h] [-v [V]] [-r REFS] [path]

Remove the "editable" mode for this reference.

positional arguments:
  path                  Path to a folder containing a recipe (conanfile.py or
                        conanfile.txt) or to a recipe file. e.g.,
                        ./my_project/conanfile.txt.

optional arguments:
  -h, --help            show this help message and exit
  -v [V]                Level of detail of the output. Valid options from less
                        verbose to more verbose: -vquiet, -verror, -vwarning,
                        -vnotice, -vstatus, -v or -vverbose, -vv or -vdebug,
                        -vvv or -vtrace
  -r REFS, --refs REFS  Directly provide reference patterns
```

See also:

- Read the tutorial about editable packages *editable package*.

7.1.16 conan export

```
$ conan export -h
usage: conan export [-h] [-f FORMAT] [-v [V]] [--name NAME]
                    [--version VERSION] [--user USER] [--channel CHANNEL]
                    [-r REMOTE | -nr] [-l LOCKFILE]
                    [--lockfile-out LOCKFILE_OUT] [--lockfile-partial]
                    [--build-require]
                    path

Export a recipe to the Conan package cache.

positional arguments:
  path                  Path to a folder containing a recipe (conanfile.py)

optional arguments:
  -h, --help            show this help message and exit
  -f FORMAT, --format FORMAT
                        Select the output format: json
  -v [V]                Level of detail of the output. Valid options from less
                        verbose to more verbose: -vquiet, -verror, -vwarning,
                        -vnotice, -vstatus, -v or -vverbose, -vv or -vdebug,
                        -vvv or -vtrace
  --name NAME           Provide a package name if not specified in conanfile
  --version VERSION     Provide a package version if not specified in
                        conanfile
  --user USER           Provide a user if not specified in conanfile
  --channel CHANNEL     Provide a channel if not specified in conanfile
  -r REMOTE, --remote REMOTE
                        Look in the specified remote or remotes server
  -nr, --no-remote      Do not use remote, resolve exclusively in the cache
  -l LOCKFILE, --lockfile LOCKFILE
                        Path to a lockfile.
  --lockfile-out LOCKFILE_OUT
                        Filename of the updated lockfile
  --lockfile-partial    Do not raise an error if some dependency is not found
                        in lockfile
  --build-require       Whether the provided reference is a build-require
```

The `conan export` command exports the recipe specified in `path` to the Conan package cache.

7.1.17 conan export-pkg

```
$ conan export-pkg -h
usage: conan export-pkg [-h] [-f FORMAT] [-v [V]]
                        [-of OUTPUT_FOLDER] [--build-require]
                        [-tf TEST_FOLDER] [--name NAME] [--version VERSION]
                        [--user USER] [--channel CHANNEL] [-l LOCKFILE]
                        [--lockfile-partial] [--lockfile-out LOCKFILE_OUT]
                        [--lockfile-packages] [--lockfile-clean]
                        [-o OPTIONS_HOST] [-o:b OPTIONS_BUILD]
                        [-o:h OPTIONS_HOST] [-pr PROFILE_HOST]
                        [-pr:b PROFILE_BUILD] [-pr:h PROFILE_HOST]
                        [-s SETTINGS_HOST] [-s:b SETTINGS_BUILD]
                        [-s:h SETTINGS_HOST] [-c CONF_HOST] [-c:b CONF_BUILD]
                        [-c:h CONF_HOST]
```

(continues on next page)

(continued from previous page)

```
                        path

Create a package directly from pre-compiled binaries.

positional arguments:
  path                  Path to a folder containing a recipe (conanfile.py)

optional arguments:
  -h, --help            show this help message and exit
  -f FORMAT, --format FORMAT
                        Select the output format: json
  -v [V]                Level of detail of the output. Valid options from less
                        verbose to more verbose: -vquiet, -verror, -vwarning,
                        -vnotice, -vstatus, -v or -vverbose, -vv or -vdebug,
                        -vvv or -vtrace
  -of OUTPUT_FOLDER, --output-folder OUTPUT_FOLDER
                        The root output folder for generated and build files
  --build-require       Whether the provided reference is a build-require
  -tf TEST_FOLDER, --test-folder TEST_FOLDER
                        Alternative test folder name. By default it is
                        "test_package". Use "" to skip the test stage
  --name NAME           Provide a package name if not specified in conanfile
  --version VERSION     Provide a package version if not specified in
                        conanfile
  --user USER           Provide a user if not specified in conanfile
  --channel CHANNEL     Provide a channel if not specified in conanfile
  -l LOCKFILE, --lockfile LOCKFILE
                        Path to a lockfile. Use --lockfile="" to avoid
                        automatic use of existing 'conan.lock' file
  --lockfile-partial    Do not raise an error if some dependency is not found
                        in lockfile
  --lockfile-out LOCKFILE_OUT
                        Filename of the updated lockfile
  --lockfile-packages   Lock package-id and package-revision information
  --lockfile-clean      Remove unused entries from the lockfile
  -o OPTIONS_HOST, --options OPTIONS_HOST
                        Define options values (host machine), e.g.: -o
                        Pkg:with_qt=true
  -o:b OPTIONS_BUILD, --options:build OPTIONS_BUILD
                        Define options values (build machine), e.g.: -o:b
                        Pkg:with_qt=true
  -o:h OPTIONS_HOST, --options:host OPTIONS_HOST
                        Define options values (host machine), e.g.: -o:h
                        Pkg:with_qt=true
  -pr PROFILE_HOST, --profile PROFILE_HOST
                        Apply the specified profile to the host machine
  -pr:b PROFILE_BUILD, --profile:build PROFILE_BUILD
                        Apply the specified profile to the build machine
  -pr:h PROFILE_HOST, --profile:host PROFILE_HOST
                        Apply the specified profile to the host machine
  -s SETTINGS_HOST, --settings SETTINGS_HOST
                        Settings to build the package, overwriting the
                        defaults (host machine). e.g.: -s compiler=gcc
  -s:b SETTINGS_BUILD, --settings:build SETTINGS_BUILD
                        Settings to build the package, overwriting the
                        defaults (build machine). e.g.: -s:b compiler=gcc
  -s:h SETTINGS_HOST, --settings:host SETTINGS_HOST
```

(continues on next page)

(continued from previous page)

```
                        Settings to build the package, overwriting the
                        defaults (host machine). e.g.: -s:h compiler=gcc
  -c CONF_HOST, --conf CONF_HOST
                        Configuration to build the package, overwriting the
                        defaults (host machine). e.g.: -c
                        tools.cmake.cmaketoolchain:generator=Xcode
  -c:b CONF_BUILD, --conf:build CONF_BUILD
                        Configuration to build the package, overwriting the
                        defaults (build machine). e.g.: -c:b
                        tools.cmake.cmaketoolchain:generator=Xcode
  -c:h CONF_HOST, --conf:host CONF_HOST
                        Configuration to build the package, overwriting the
                        defaults (host machine). e.g.: -c:h
                        tools.cmake.cmaketoolchain:generator=Xcode
```

Warning: The json output of the `conan export-pkg --format=json` is **experimental** and subject to change.

See also:

- Read the tutorial about the *local package developement flow*.

7.1.18 conan new

Create a new recipe (with a conanfile.py and other associated files) from either a predefined or a user-defined template.

conan new

```
$ conan new -h
usage: conan new [-h] [-v [V]] [-d DEFINE] [-f] template

Create a new example recipe and source files from a template.

positional arguments:
  template              Template name, either a predefined built-in or a user-
                        provided one. Available built-in templates: basic,
                        cmake_lib, cmake_exe, meson_lib, meson_exe,
                        msbuild_lib, msbuild_exe, bazel_lib, bazel_exe,
                        autotools_lib, autotools_exe. E.g. 'conan new
                        cmake_lib -d name=hello -d version=0.1'. You can
                        define your own templates too by inputting an absolute
                        path as your template, or a path relative to your
                        conan home folder.

optional arguments:
  -h, --help            show this help message and exit
  -v [V]                Level of detail of the output. Valid options from less
                        verbose to more verbose: -vquiet, -verror, -vwarning,
                        -vnotice, -vstatus, -v or -vverbose, -vv or -vdebug,
                        -vvv or -vtrace
  -d DEFINE, --define DEFINE
                        Define a template argument as key=value, e.g., -d
```

(continues on next page)

(continued from previous page)

```
                        name=mypkg
  -f, --force           Overwrite file if it already exists
```

The `conan new` command creates a new recipe in the current working directory, plus extra example files such as *CMakeLists.txt* or the *test_package* folder (as necessary), to either be used as a basis for your own project or aiding in the debugging process.

Note that each template has some required and some [optional] user-defined variables used to customize the resulting files.

The available templates are:

- **basic**: Creates a simple recipe with some example code and helpful comments, and is a good starting point to avoid writing boilerplate code.

  Its variables are: [name], [version], [description], [require, require…]

- **alias**: Creates the minimal recipe needed to define an alias to a target recipe

  Its variables are: name, [version], target

- **cmake_lib**: Creates a cmake library target that defines a function called `name`, which will print some information about the compilation environment to stdout. You can add requirements to this template in the form of

  `conan new cmake_lib -d name=ai -d version=1.0 -d requires=math/3.14 -d requires=magic/0.0`

  This will add requirements for both `math/3.14` and `magic/0.0` to the *requirements()* method, will add the necessary `find_package``s in CMake, and add a call to ``math()` and `magic()` inside the generated `ai()` function.

  Its variables are: name, version, [require, require…]

- **cmake_exe**: Creates a cmake executable target that defines a function called `name`, which will print some information about the compilation environment to stdout. You can add requirements to this template in the form of

  `conan new cmake_exe -d name=game -d version=1.0 -d requires=math/3.14 -d requires=ai/1.0`

  This will add requirements for both `math/3.14` and `ai/1.0` to the *requirements()* method, will add the necessary `find_package``s in CMake, and add a call to ``math()` and `ai()` inside the generated `game()` function.

  Its variables are: name, version, [require, require…]

- **autotools_lib**: Creates an Autotools library.

  Its variables are: `name`, `version`

- **autotools_exe**: Creates an Autotools executable

  Its variables are: `name`, `version`

- **bazel_lib**: **Bazel integration BazelDeps, BazelToolchain, Bazel is experimental**. Creates a Bazel library.

  Its variables are: `name`, `version`

- **bazel_exe**: **Bazel integration BazelDeps, BazelToolchain, Bazel is experimental**. Creates a Bazel executable

  Its variables are: `name`, `version`

- **meson_lib**: Creates a Meson library.

  Its variables are: `name`, `version`

- **meson_exe**: Creates a Meson executable

  Its variables are: `name`, `version`

- **msbuild_lib**: Creates a MSBuild library.

  Its variables are: `name`, `version`

- **msbuild_exe**: Creates a MSBuild executable

  Its variables are: `name`, `version`

> **Warning:** The output of the predefined built-in templates is **not stable**. It might change in future releases to adapt to the latest tools or good practices.

Examples

```
$ conan new basic
```

Generates a basic *conanfile.py* that does not implement any custom functionality

```
$ conan new basic -d name=mygame -d requires=math/1.0 -d requires=ai/1.3
```

Generates a *conanfile.py* for `mygame` that depends on the packages `math/1.0` and `ai/1.3`

```
$ conan new cmake_exe -d name=game -d version=1.0 -d requires=math/3.14 -d␣
↪requires=ai/1.0
```

Generates the necessary files for a CMake executable target. This will add requirements for both `math/3.14` and `ai/1.0` to the `requirements()` method, will add the necessary `find_package` in CMake, and add a call to `math()` and `ai()` inside the generated `game()` function.

Custom templates

There's also the possibility to create your own templates by passing a path to your template directory, both as an absolute path, or relative to your Conan home folder. This directory should contain Jinja2 templates, which will produce your desired template structure. You can use custom variables that will be needed to be passed as `name` and `` `version `` does, or use your custom variables.

7.1.19 conan source

```
$ conan source -h
usage: conan source [-h] [-v [V]] [--name NAME] [--version VERSION]
                    [--user USER] [--channel CHANNEL]
                    [path]

Call the source() method.

positional arguments:
  path                  Path to a folder containing a recipe (conanfile.py or
```

(continues on next page)

(continued from previous page)

```
                        conanfile.txt) or to a recipe file. e.g.,
                        ./my_project/conanfile.txt.

optional arguments:
  -h, --help            show this help message and exit
  -v [V]                Level of detail of the output. Valid options from less
                        verbose to more verbose: -vquiet, -verror, -vwarning,
                        -vnotice, -vstatus, -v or -vverbose, -vv or -vdebug, -vvv
                        or -vtrace
  --name NAME           Provide a package name if not specified in conanfile
  --version VERSION     Provide a package version if not specified in conanfile
  --user USER           Provide a user if not specified in conanfile
  --channel CHANNEL     Provide a channel if not specified in conanfile
```

See also:

- Read the tutorial about the *local package developement flow*.

7.1.20 conan test

```
$ conan test -h
usage: conan test [-h] [-v [V]] [-b BUILD] [-r REMOTE | -nr] [-u]
                  [-o OPTIONS_HOST] [-o:b OPTIONS_BUILD] [-o:h OPTIONS_HOST]
                  [-pr PROFILE_HOST] [-pr:b PROFILE_BUILD]
                  [-pr:h PROFILE_HOST] [-s SETTINGS_HOST]
                  [-s:b SETTINGS_BUILD] [-s:h SETTINGS_HOST] [-c CONF_HOST]
                  [-c:b CONF_BUILD] [-c:h CONF_HOST] [-l LOCKFILE]
                  [--lockfile-partial] [--lockfile-out LOCKFILE_OUT]
                  [--lockfile-packages] [--lockfile-clean]
                  path reference

Test a package from a test_package folder.

positional arguments:
  path                  Path to a test_package folder containing a
                        conanfile.py
  reference             Provide a package reference to test

optional arguments:
  -h, --help            show this help message and exit
  -v [V]                Level of detail of the output. Valid options from less
                        verbose to more verbose: -vquiet, -verror, -vwarning,
                        -vnotice, -vstatus, -v or -vverbose, -vv or -vdebug,
                        -vvv or -vtrace
  -b BUILD, --build BUILD
                        Optional, specify which packages to build from source.
                        Combining multiple '--build' options on one command
                        line is allowed. Possible values: --build="*" Force
                        build from source for all packages. --build=never
                        Disallow build for all packages, use binary packages
                        or fail if a binary package is not found. Cannot be
                        combined with other '--build' options. --build=missing
                        Build packages from source whose binary package is not
                        found. --build=cascade Build packages from source that
                        have at least one dependency being built from source.
```

(continues on next page)

(continued from previous page)

```
                        --build=[pattern] Build packages from source whose
                        package reference matches the pattern. The pattern
                        uses 'fnmatch' style wildcards. --build=![pattern]
                        Excluded packages, which will not be built from the
                        source, whose package reference matches the pattern.
                        The pattern uses 'fnmatch' style wildcards.
                        --build=missing:[pattern] Build from source if a
                        compatible binary does not exist, only for packages
                        matching pattern.
  -r REMOTE, --remote REMOTE
                        Look in the specified remote or remotes server
  -nr, --no-remote      Do not use remote, resolve exclusively in the cache
  -u, --update          Will check the remote and in case a newer version
                        and/or revision of the dependencies exists there, it
                        will install those in the local cache. When using
                        version ranges, it will install the latest version
                        that satisfies the range. Also, if using revisions, it
                        will update to the latest revision for the resolved
                        version range.
  -o OPTIONS_HOST, --options OPTIONS_HOST
                        Define options values (host machine), e.g.: -o
                        Pkg:with_qt=true
  -o:b OPTIONS_BUILD, --options:build OPTIONS_BUILD
                        Define options values (build machine), e.g.: -o:b
                        Pkg:with_qt=true
  -o:h OPTIONS_HOST, --options:host OPTIONS_HOST
                        Define options values (host machine), e.g.: -o:h
                        Pkg:with_qt=true
  -pr PROFILE_HOST, --profile PROFILE_HOST
                        Apply the specified profile to the host machine
  -pr:b PROFILE_BUILD, --profile:build PROFILE_BUILD
                        Apply the specified profile to the build machine
  -pr:h PROFILE_HOST, --profile:host PROFILE_HOST
                        Apply the specified profile to the host machine
  -s SETTINGS_HOST, --settings SETTINGS_HOST
                        Settings to build the package, overwriting the
                        defaults (host machine). e.g.: -s compiler=gcc
  -s:b SETTINGS_BUILD, --settings:build SETTINGS_BUILD
                        Settings to build the package, overwriting the
                        defaults (build machine). e.g.: -s:b compiler=gcc
  -s:h SETTINGS_HOST, --settings:host SETTINGS_HOST
                        Settings to build the package, overwriting the
                        defaults (host machine). e.g.: -s:h compiler=gcc
  -c CONF_HOST, --conf CONF_HOST
                        Configuration to build the package, overwriting the
                        defaults (host machine). e.g.: -c
                        tools.cmake.cmaketoolchain:generator=Xcode
  -c:b CONF_BUILD, --conf:build CONF_BUILD
                        Configuration to build the package, overwriting the
                        defaults (build machine). e.g.: -c:b
                        tools.cmake.cmaketoolchain:generator=Xcode
  -c:h CONF_HOST, --conf:host CONF_HOST
                        Configuration to build the package, overwriting the
                        defaults (host machine). e.g.: -c:h
                        tools.cmake.cmaketoolchain:generator=Xcode
  -l LOCKFILE, --lockfile LOCKFILE
                        Path to a lockfile. Use --lockfile="" to avoid
```

(continues on next page)

(continued from previous page)

```
                        automatic use of existing 'conan.lock' file
  --lockfile-partial    Do not raise an error if some dependency is not found
                        in lockfile
  --lockfile-out LOCKFILE_OUT
                        Filename of the updated lockfile
  --lockfile-packages   Lock package-id and package-revision information
  --lockfile-clean      Remove unused entries from the lockfile
```

The `conan test` command uses the *test_package* folder specified in `path` to tests the package reference specified in `reference`.

See also:

- Read the tutorial about *testing Conan packages*.

7.1.21 conan upload

Use this command to upload recipes and binaries to Conan repositories. For more information on how to work with Conan repositories, please check the *dedicated section*.

```
$ conan upload -h
usage: conan upload [-h] [-v [V]] [-p PACKAGE_QUERY] -r REMOTE
                    [--only-recipe] [--force] [--check] [-c]
                    reference

Upload packages to a remote.

By default, all the matching references are uploaded (all revisions).
By default, if a recipe reference is specified, it will upload all the revisions for␣
↪all the
binary packages, unless --only-recipe is specified. You can use the "latest"␣
↪placeholder at the
"reference" argument to specify the latest revision of the recipe or the package.

positional arguments:
  reference             Recipe reference or package reference, can contain *
                        as wildcard at any reference field. If no revision is
                        specified, it is assumed to be the latest

optional arguments:
  -h, --help            show this help message and exit
  -v [V]                Level of detail of the output. Valid options from less
                        verbose to more verbose: -vquiet, -verror, -vwarning,
                        -vnotice, -vstatus, -v or -vverbose, -vv or -vdebug,
                        -vvv or -vtrace
  -p PACKAGE_QUERY, --package-query PACKAGE_QUERY
                        Only upload packages matching a specific query. e.g:
                        os=Windows AND (arch=x86 OR compiler=gcc)
  -r REMOTE, --remote REMOTE
                        Upload to this specific remote
  --only-recipe         Upload only the recipe/s, not the binary packages.
  --force               Force the upload of the artifacts even if the revision
                        already exists in the server
  --check               Perform an integrity check, using the manifests,
                        before upload
  -c, --confirm         Upload all matching recipes without confirmation
```

Read more

- *Uploading packages tutorial*
- *Working with Conan repositories*
- *Managing remotes with conan remote command*
- *conan build*: Install package and call its build method
- *conan create*: Create a package from a recipe
- *conan download*: Download (without install) a single conan package from a remote server.
- *conan editable*: Allows working with a package in user folder
- *conan export*: Export a recipe to the Conan package cache
- *conan export-pkg*: Create a package directly from pre-compiled binaries
- *conan new*: Create a new recipe from a predefined template
- *conan source*: Calls the source() method
- *conan test*: Test a package
- *conan upload*: Upload packages from the local cache to a specified remote

7.2 conanfile.py

The `conanfile.py` is the recipe file of a package, responsible for defining how to build it and consume it.

```
from conan import ConanFile

class HelloConan(ConanFile):
    ...
```

Important: *conanfile.py* recipes uses a variety of attributes and methods to operate. In order to avoid collisions and conflicts, follow these rules:

- Public attributes and methods, like `build()`, `self.package_folder`, are reserved for Conan. Don't use public members for custom fields or methods in the recipes.
- Use "protected" access for your own members, like `self._my_data` or `def _my_helper(self):`. Conan only reserves "protected" members starting with `_conan`.

Contents:

7.2.1 Attributes

Package reference

Recipe attributes that can define the main `pkg/version@user/channel` package reference.

name

The name of the package. A valid name is all lowercase and has:

- A minimum of 2 and a maximum of 101 characters (though shorter names are recommended).
- **Matches the following regex `^[a-z0-9_][a-z0-9_+.-]{1,100}$`: so starts with alphanumeric or _,** then from 1 to 100 characters between alphanumeric, _, +, . or -.

The name is only necessary for `export`-ing the recipe into the local cache (`export`, `export-pkg` and `create` commands), if they are not defined in the command line with `--name=<pkgname>`.

version

The version of the package. A valid version follows the same rules than 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`, `export-pkg` and `create` commands), if they are not defined in the command line with `--version=<pkgversion>`

The `version` can be dynamically defined in the command line, and also programmaticaly in the recipe with the *set_version() method*.

user

A valid string for the `user` field follows the same rules than the `name` attribute. This is an optional attribute. It can be used to identify your own packages with `pkg/version@user/channel`, where `user` could be the name of your team, org or company. ConanCenter recipes don't have `user/channel`, so they are in the form of `pkg/version` only. You can also name your packages without user and channel, or using only the user as `pkg/version@user`.

The user can be specified in the command line with `--user=<myuser>`

channel

A valid string for the `channel` field follows the same rules than the `name` attribute. This is an optional attribute. It is sometimes used to identify a maturity of the package ("stable", "testing"...), but in general this is not necessary, and the maturity of packages is better managed by putting them in different server repositories.

The user can be specified in the command line with `--channel=<mychannel>`

Metadata

Optional metadata, like license, description, author, etc. Not necessary for most cases, but can be useful to have.

description

This is an optional, but 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"""
```

license

License of the **target** source code and binaries, i.e. the code that is being packaged, not the `conanfile.py` itself. Can contain several, comma separated licenses. It is a text string, so it can contain any text, but it is strongly recommended that recipes of Open Source projects use SPDX identifiers from the SPDX license list

This will help people wanting to automate license compatibility checks, like consumers of your package, or you if your package has Open-Source dependencies.

```
class Pkg(ConanFile):
    license = "MIT"
```

author

Main maintainer/responsible for the package, any format. This is an optional attribute.

```
class HelloConan(ConanFile):
    author = "John J. Smith (john.smith@company.com)"
```

topics

Tags to group related packages together and describe what the code is about. Used as a search filter in ConanCenter. Optional attribute. It should be a tuple of strings.

```
class ProtocInstallerConan(ConanFile):
    name = "protoc_installer"
    version = "0.1"
    topics = ("protocol-buffers", "protocol-compiler", "serialization", "rpc")
```

homepage

The home web page of the library being packaged.

Used 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

URL of the package repository, i.e. not necessarily of the original source code. Recommended, but not mandatory attribute.

```
class HelloConan(ConanFile):
    name = "hello"
    version = "0.1"
    url = "https://github.com/conan-io/hello.git"
```

Requirements

Attribute form of the dependencies simple declarations, like `requires`, `tool_requires`. For more advanced way to define requirements, use the `requirements()`, `build_requirements()` methods instead.

requires

List or tuple of strings for regular dependencies in the host context, like a library.

```python
class MyLibConan(ConanFile):
    requires = "hello/1.0", "otherlib/2.1@otheruser/testing"
```

You can specify version ranges, the syntax is using brackets:

```python
class HelloConan(ConanFile):
    requires = "pkg/[>1.0 <1.8]"
```

Accepted expressions would be:

Expression	Versions in range	Versions outside of range
[>=1.0 <2]	1.0.0, 1.0.1, 1.1, 1.2.3	0.2, 2.0, 2.1, 3.0
[<3.2.1]	0.1, 1.2, 2.4, 3.1.1	3.2.2
[>2.0]	2.1, 2.2, 3.1, 14.2	1.1, 1.2, 2.0

See also:

- Check <MISSING PAGE> version_ranges if you want to learn more about version ranges.
- Check <MISSING PAGE> requires() conanfile.py method.

tool_requires

List or tuple of strings for dependencies. Represents a build tool like "cmake". If there is an existing pre-compiled binary for the current package, the binaries for the tool_require won't be retrieved. They cannot conflict.

```python
class MyPkg(ConanFile):
    tool_requires = "tool_a/0.2", "tool_b/0.2@user/testing"
```

This is the declarative way to add `tool_requires`. Check the <MISSING PAGE> tool_requires() conanfile.py method to learn a more flexible way to add them.

build_requires

List or tuple of strings for dependencies. Generic type of build dependencies that are not applications (nothing runs), like build scripts. If there is an existing pre-compiled binary for the current package, the binaries for the build_require won't be retrieved. They cannot conflict.

```python
class MyPkg(ConanFile):
    build_requires = ["my_build_scripts/1.3",]
```

This is the declarative way to add `build_requires`. Check the <MISSING PAGE> build_requires() conanfile.py method to learn a more flexible way to add them.

test_requires

List or tuple of strings for dependencies in the host context only. Represents a test tool like "gtest". Used when the current package is built from sources. They don't propagate information to the downstream consumers. If there is an existing pre-compiled binary for the current package, the binaries for the test_require won't be retrieved. They cannot conflict.

```
class MyPkg(ConanFile):
    test_requires = "gtest/1.11.0", "other_test_tool/0.2@user/testing"
```

This is the declarative way to add `test_requires`. Check the <MISSING PAGE> test_requires() conanfile.py method to learn a more flexible way to add them.

python_requires

This class attribute allows to define a dependency to another Conan recipe and reuse its code. Its basic syntax is:

```
from conan import ConanFile

class Pkg(ConanFile):
    python_requires = "pyreq/0.1@user/channel"  # recipe to reuse code from

    def build(self):
        self.python_requires["pyreq"].module # access to the whole conanfile.py module
        self.python_requires["pyreq"].module.myvar  # access to a variable
        self.python_requires["pyreq"].module.myfunct()  # access to a global function
        self.python_requires["pyreq"].path # access to the folder where the reused␣
↪file is
```

Read more about this attribute in *Python requires*

python_requires_extend

This class attribute defines one or more classes that will be injected in runtime as base classes of the recipe class. Syntax for each of these classes should be a string like `pyreq.MyConanfileBase` where the `pyreq` is the name of a `python_requires` and `MyConanfileBase` is the name of the class to use.

```
from conan import ConanFile

class Pkg(ConanFile):
    python_requires = "pyreq/0.1@user/channel", "utils/0.1@user/channel"
    python_requires_extend = "pyreq.MyConanfileBase", "utils.UtilsBase"  # class/es␣
↪to inject
```

Sources

exports

List or tuple of strings with *file names* or fnmatch patterns that should be exported and stored side by side with the *conanfile.py* file to make the recipe work: other python files that the recipe will import, some text file with data to read,. . .

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"
```

See also:

- Check <MISSING PAGE> exports() conanfile.py method.

exports_sources

List or tuple of strings with file names or fnmatch patterns that should be exported and will be available to generate the package. Unlike the `exports` attribute, these files shouldn't be used by the `conanfile.py` Python code, but to compile the library or generate the final package. And, due to its purpose, these files will only be retrieved if requested binaries are not available or the user forces Conan to compile from sources.

This is an alternative to getting the sources with the `source()` method. Used when we are not packaging a third party library and we have together the recipe and the C/C++ project:

```
exports_sources = "include*", "src*"
```

Exclude patterns are also possible, with the `!` prefix:

```
exports_sources = "include*", "src*", "!src/build/*"
```

Note, if the recipe defines the `layout()` method and specifies a `self.folders.source = "src"` it won't affect where the files (from the `exports_sources`) are copied. They will be copied to the base source folder. So, if you want to replace some file that got into the `source()` method, you need to explicitly copy it from the parent folder or even better, from `self.export_sources_folder`.

```
import os, shutil
from conan import ConanFile
from conan.tools.files import save, load

class Pkg(ConanFile):
    ...
    exports_sources = "CMakeLists.txt"

    def layout(self):
        self.folders.source = "src"
        self.folders.build = "build"

    def source(self):
        # emulate a download from web site
        save(self, "CMakeLists.txt", "MISTAKE: Very old CMakeLists to be replaced")
        # Now I fix it with one of the exported files
        shutil.copy("../CMakeLists.txt", ".")
        shutil.copy(os.path.join(self.export_sources_folder, "CMakeLists.txt", "."))
```

conan_data

Read only attribute with a dictionary with the keys and values provided in a <MISSING PAGE> conandata_yml file format placed next to the *conanfile.py*. This YAML file is automatically exported with the recipe and automatically loaded with it too.

You can declare information in the *conandata.yml* file and then access it inside any of the methods of the recipe. For example, a *conandata.yml* with information about sources that looks like this:

```yaml
sources:
  "1.1.0":
    url: "https://www.url.org/source/mylib-1.0.0.tar.gz"
    sha256: "8c48baf3babe0d505d16cfc0cf272589c66d3624264098213db0fb00034728e9"
  "1.1.1":
    url: "https://www.url.org/source/mylib-1.0.1.tar.gz"
    sha256: "15b6393c20030aab02c8e2fe0243cb1d1d18062f6c095d67bca91871dc7f324a"
```

```python
def source(self):
    tools.get(**self.conan_data["sources"][self.version])
```

Binary model

Important attributes that define the package binaries model, which settings, options, package type, etc. affect the final packaged binaries.

package_type

Optional. Declaring the `package_type` will help Conan:

- To choose better the default `package_id_mode` for each dependency, that is, how a change in a dependency should affect the `package_id` to the current package.
- Which information from the dependencies should be propagated to the consumers, like headers, libraries, runtime information…

The valid values are:

- **application**: The package is an application.
- **library**: The package is a generic library. It will try to determine the type of library (from `shared-library`, `static-library`, `header-library`) reading the `self.options.shared` (if declared) and the `self.options.header_only`
- **shared-library**: The package is a shared library.
- **static-library**: The package is a static library.
- **header-library**: The package is a header only library.
- **build-scripts**: The package only contains build scripts.
- **python-require**: The package is a python require.
- **unknown**: The type of the package is unknown.

settings

List of strings with the first level settings (from `settings.yml`) that the recipe needs, because: - They are read for building (e.g: *if self.settings.compiler == "gcc"*) - They affect the `package_id`. If a value of the declared setting changes, the `package_id` has to be different.

The most common is to declare:

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

Once the recipe is loaded by Conan, the `settings` are processed and they can be read in the recipe, also the sub-settings:

```
settings = "os", "arch"

def build(self):
    if self.settings.compiler == "gcc":
        if self.settings.compiler.cppstd == "gnu20":
            # do some special build commands
```

If you try to access some setting that doesn't exist, like `self.settings.compiler.libcxx` for the `msvc` setting, Conan will fail telling that `libcxx` does not exist for that compiler.

If you want to do a safe check of settings values, you could use the `get_safe()` method:

```
def build(self):
    # Will be None if doesn't exist (not declared)
    arch = self.settings.get_safe("arch")
    # Will be None if doesn't exist (doesn't exist for the current compiler)
    compiler_version = self.settings.get_safe("compiler.version")
    # Will be the default version if the return is None
    build_type = self.settings.get_safe("build_type", default="Release")
```

The `get_safe()` method will return `None` if that setting or sub-setting doesn't exist and there is no default value assigned.

If you want to do a safe deletion of settings, you could use the `rm_safe()` method. For example, in the `configure()` method a typical pattern for a C library would be:

```
def configure(self):
    self.settings.rm_safe("compiler.libcxx")
    self.settings.rm_safe("compiler.cppstd")
```

See also:

- Removing settings in the `package_id()` method. <MISSING PAGE>

options

Dictionary with traits that affects only the current recipe, where the key is the option name and the value is a list of different values that the option can take. By default any value change in an option, changes the `package_id`. Check the `default_options` field to define default values for the options.

Values for each option can be typed or plain strings (`"value"`, `True`, `42`,`...`).

There are two special values:

- `None`: Allow the option to have a `None` value (not specified) without erroring.
- `"ANY"`: For options that can take any value, not restricted to a set.

```
class MyPkg(ConanFile):
    ...
    options = {
        "shared": [True, False],
        "option1": ["value1", "value2"],
        "option2": ["ANY"],
        "option3": [None, "value1", "value2"],
```

(continues on next page)

(continued from previous page)

```
        "option4": [True, False, "value"],
}
```

Once the recipe is loaded by Conan, the `options` are processed and they can be read in the recipe. You can also use the method `.get_safe()` (see *settings attribute*) to avoid Conan raising an Exception if the option doesn't exist:

```
class MyPkg(ConanFile):
    options = {"shared": [True, False]}

    def build(self):
        if self.options.shared:
            # build the shared library
        if self.options.get_safe("foo", True):
            pass
```

In boolean expressions, like `if self.options.shared`:

- equals `True` for the values `True`, `"True"` and `"true"`, and any other value that would be evaluated the same way in Python code.
- equals `False` for the values `False`, `"False"` and `"false"`, also for the empty string and for `0` and `"0"` as expected.

Notice that a comparison using `is` is always `False` because the types would be different as it is encapsulated inside a Python class.

If you want to do a safe deletion of options, you could use the `rm_safe()` method. For example, in the `config_options()` method a typical pattern for Windows library would be:

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

See also:

- Read the *Getting started, creating packages* to know how to declare and how to define a value to an option.
- Removing options in the `package_id()` method. <MISSING PAGE>
- About the package_type and how it plays when a `shared` option is declared. <MISSING PAGE>

default_options

The attribute `default_options` defines the default values for the options, both for the current recipe and for any requirement. This attribute should be defined as a python dictionary.

```
class MyPkg(ConanFile):
    ...
    requires = "zlib/1.2.8", "zwave/2.0"
    options = {"build_tests": [True, False],
               "option2": "ANY"}
    default_options = {"build_tests": True,
                       "option1": 42,
                       "z*: shared": True}
```

You can also assign default values for options of your requirements using "<reference_pattern>: option_name", being a valid `reference_pattern` a `name/version` or any pattern with `*` like the example above.

You can also set the options conditionally to a final value with `configure()` instead of using `default_options`:

```
class OtherPkg(ConanFile):
    settings = "os", "arch", "compiler", "build_type"
    options = {"some_option": [True, False]}
    # Do NOT declare 'default_options', use 'config_options()'

    def configure(self):
        if self.options.some_option == None:
            if self.settings.os == 'Android':
                self.options.some_option = True
            else:
                self.options.some_option = False
```

Take into account that if a value is assigned in the `configure()` method it cannot be overridden.

See also:

Read more about the <MISSING PAGE>method_configure_config_options method.

options_description

TODO: Complete, https://github.com/conan-io/conan/pull/11295

info

Object used exclusively in `package_id()` method:

- The <MISSING PAGE> `package_id(self)` method to control the unique ID for a package:

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

Build

generators

List or tuple of strings with names of generators.

```
class MyLibConan(ConanFile):
    generators = "CMakeDeps", "CMakeToolchain"
```

The generators can also be instantiated explicitly in the <MISSING PAGE> generate() method.

```
from conan.tools.cmake import CMakeToolchain

class MyLibConan(ConanFile):
    ...

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

build_policy

Controls when the current package is built during a `conan install`. The allowed values are:

- `"missing"`: Conan builds it from source if there is no binary available.
- `"never"`: This package cannot be built from sources, it is always created with `conan export-pkg`
- `None` (default value): This package won't be built unless the policy is specified in the command line (e.g `--build=foo*`)

```
class PocoTimerConan(ConanFile):
    build_policy = "missing"
```

win_bash

When `True` it enables the new run in a subsystem bash in Windows mechanism.

```
from conan import ConanFile

class FooRecipe(ConanFile):
    ...
    win_bash = True
```

It can also be declared as a `property` based on any condition:

```
from conan import ConanFile

class FooRecipe(ConanFile):
    ...


    @property
    def win_bash(self):
        return self.settings.arch == "armv8"
```

win_bash_run

When `True` it enables running commands in the `"run"` scope, to run them inside a bash shell.

```
from conan import ConanFile

class FooRecipe(ConanFile):

    ...

    win_bash_run = True
    def build(self):
        self.run(cmd, scope="run")  # will run <cmd> inside bash
```

Folders and layout

source_folder

The folder in which the source code lives. The path is built joining the base directory (a cache directory when running in the cache or the `output folder` when running locally) with the value of `folders.source` if declared in the `layout()` method.

Note that the base directory for the `source_folder` when running in the cache will point to the base folder of the build unless *no_copy_source* is set to `True`. But anyway it will always point to the correct folder where the source code is.

export_sources_folder

The value depends on the method you access it:

- At `source(self)`: Points to the base source folder (that means self.source_folder but without taking into account the `folders.source` declared in the `layout()` method). The declared *exports_sources* are copied to that base source folder always.
- At `exports_sources(self)`: Points to the folder in the cache where the export sources have to be copied.

See also:

- Read <MISSING PAGE> `export_sources` method.
- Read <MISSING PAGE> `source` method.

build_folder

The folder used to build the source code. The path is built joining the base directory (a cache directory when running in the cache or the `output folder` when running locally) with the value of `folders.build` if declared in the `layout()` method.

package_folder

The folder to copy the final artifacts for the binary package. In the local cache a package folder is created for every different package ID.

The most common usage of `self.package_folder` is to `copy` the files at the <MISSING PAGE> package() method:

```
import os
from conan import ConanFile
from conan.tools.files import copy

class MyRecipe(ConanFile):
    ...

    def package(self):
        copy(self, "*.so", self.build_folder, os.path.join(self.package_folder, "lib
↪"))
        ...
```

recipe_folder

The folder where the recipe *conanfile.py* is stored, either in the local folder or in the cache. This is useful in order to access files that are exported along with the recipe, or the origin folder when exporting files in `export(self)` and `export_sources(self)` methods.

The most common usage of `self.recipe_folder` is in the `export(self)` and `export_sources(self)` methods, as the folder from where we copy the files:

```
from conan import ConanFile
from conan.tools.files import copy

class MethodConan(ConanFile):
    exports = "file.txt"
    def export(self):
        copy(self, "LICENSE.md", self.recipe_folder, self.export_folder)
```

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 or header-only, to avoid extra copies.

If you activate `no_copy_source=True`, 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.

The recipes should always use `self.source_folder` attribute, which will point to the `build` folder when `no_copy_source=False` and will point to the `source` folder when `no_copy_source=True`.

See also:

Read <MISSING PAGE> header-only section for an example using `no_copy_source` attribute.

Layout

folders

The `folders` attribute has to be set only in the `layout()` method. Please check the *layout() method documentation* to learn more about this attribute.

cpp

Object storing all the information needed by the consumers of a package: include directories, library names, library paths… Both for editable and regular packages in the cache. It is only available at the `layout()` method.

- `self.cpp.package`: For a regular package being used from the Conan cache. Same as declaring `self.cpp_info` at the `package_info()` method.
- `self.cpp.source`: For "editable" packages, to describe the artifacts under `self.source_folder`
- `self.cpp.build`: For "editable" packages, to describe the artifacts under `self.build_folder`.

The `cpp` attribute has to be set only in the `layout()` method. Please check the *layout() method documentation* to learn more about this attribute.

layouts

The `layouts` attribute has to be set only in the `layout()` method. Please check the *layout() method documentation* to learn more about this attribute.

The `layouts` attribute contains information about environment variables and `conf` that would be path-dependent, and as a result it would contain a different value when the package is in editable mode, or when the package is in the cache. The `layouts` sub-attributes are:

- `self.layouts.build`: information related to the relative `self.folders.build`
- `self.layouts.source`: information related to the relative `self.folders.source`
- `self.layouts.package`: information related to the final `package_folder`

Each one of those will contain:

- `buildenv_info`: environment variables build information for consumers (equivalent to `self.buildenv_info` in `package_info()`)
- `runenv_info`: environment variables run information for consumers (equivalent to `self.runenv_info` in `package_info()`)
- `conf_info`: configuration information for consumers (equivalent to `self.conf_info` in `package_info()`). Note this is only automatically propagated to `self.conf` of consumers when this package is a direct `tool_require`.

For example, if we had an `androidndk` recipe that contais the AndroidNDK, and we want to have that recipe in "editable" mode, it is necessary where the androidndk will be locally, before being in the created package:

```
import os
from conan import ConanFile
from conan.tools.files import copy

class AndroidNDK(ConanFile):

    def layout(self):
        # When developing in user space it is in a "mybuild" folder (relative to␣
↪current dir)
        self.layouts.build.conf_info.define_path("tools.android:ndk_path", "mybuild")
        # but when packaged it will be in a "mypkg" folder (inside the cache package␣
↪folder)
        self.layouts.package.conf_info.define_path("tools.android:ndk_path", "mypkg")

    def package(self):
        copy(self, "*", src=os.path.join(self.build_folder, "mybuild"),
             dst=os.path.join(self.package_folder, "mypkg"))
```

Package information for consumers

cpp_info

Same as using `self.cpp.package` in the `layout()` method. Use it if you need to read the `package_folder` to locate the already located artifacts.

See also:

Read more about the *CppInfo* model.

Important: This attribute is only defined inside `package_info()` method being *None* elsewhere.

buildenv_info

For the dependant recipes, the declared environment variables will be present during the build process. Should be only filled in the `package_info()` method.

Important: This attribute is only defined inside `package_info()` method being *None* elsewhere.

```
def package_info(self):
    self.buildenv_info.append_path("PATH", self.package_folder)
```

See also:

Check the reference of the *Environment* object to know how to fill the `self.buildenv_info`.

runenv_info

For the dependant recipes, the declared environment variables will be present at runtime. Should be only filled in the `package_info()` method.

Important: This attribute is only defined inside `package_info()` method being *None* elsewhere.

```
def package_info(self):
    self.runenv_info.define_path("RUNTIME_VAR", "c:/path/to/exe")
```

See also:

Check the reference of the *Environment* object to know how to fill the `self.runenv_info`.

conf_info

Configuration variables to be passed to the dependant recipes. Should be only filled in the `package_info()` method.

```
class Pkg(ConanFile):
    name = "pkg"

    def package_info(self):
        self.conf_info.define("tools.microsoft.msbuild:verbosity", "Diagnostic")
        self.conf_info.get("tools.microsoft.msbuild:verbosity")  # == "Diagnostic"
        self.conf_info.append("user.myconf.build:ldflags", "--flag3")  # == ["--flag1
↪", "--flag2", "--flag3"]
        self.conf_info.update("tools.microsoft.msbuildtoolchain:compile_options", {
↪"ExpandAttributedSource": "false"})
        self.conf_info.unset("tools.microsoft.msbuildtoolchain:compile_options")
        self.conf_info.remove("user.myconf.build:ldflags", "--flag1")  # == ["--flag0
↪", "--flag2", "--flag3"]
        self.conf_info.pop("tools.system.package_manager:sudo")
```

See also:

Read here *the complete reference of self.conf_info*.

deprecated

This attribute declares that the recipe is deprecated, causing a user-friendly warning message to be emitted whenever it is used

For example, the following code:

```
from conan import ConanFile

class Pkg(ConanFile):
    name = "cpp-taskflow"
    version = "1.0"
    deprecated = True
```

may emit a warning like:

```
cpp-taskflow/1.0: WARN: Recipe 'cpp-taskflow/1.0' is deprecated. Please, consider␣
↪changing your requirements.
```

Optionally, the attribute may specify the name of the suggested replacement:

```
from conan import ConanFile

class Pkg(ConanFile):
    name = "cpp-taskflow"
    version = "1.0"
    deprecated = "taskflow"
```

This will emit a warning like:

```
cpp-taskflow/1.0: WARN: Recipe 'cpp-taskflow/1.0' is deprecated in favor of 'taskflow
↪'. Please, consider changing your requirements.
```

If the value of the attribute evaluates to `False`, no warning is printed.

provides

This attribute declares that the recipe provides the same functionality as other recipe(s). The attribute is usually needed if two or more libraries implement the same API to prevent link-time and run-time conflicts (ODR violations). One typical situation is forked libraries. Some examples are:

- LibreSSL, BoringSSL and OpenSSL
- libav and ffmpeg
- MariaDB client and MySQL client

If Conan encounters two or more libraries providing the same functionality within a single graph, it raises an error:

```
At least two recipes provides the same functionality:
- 'libjpeg' provided by 'libjpeg/9d', 'libjpeg-turbo/2.0.5'
```

The attribute value should be a string with a recipe name or a tuple of such recipe names.

For example, to declare that `libjpeg-turbo` recipe offers the same functionality as `libjpeg` recipe, the following code could be used:

```
from conan import ConanFile

class LibJpegTurbo(ConanFile):
    name = "libjpeg-turbo"
    version = "1.0"
    provides = "libjpeg"
```

To declare that a recipe provides the functionality of several different recipes at the same time, the following code could be used:

```
from conan import ConanFile

class OpenBLAS(ConanFile):
    name = "openblas"
    version = "1.0"
    provides = "cblas", "lapack"
```

If the attribute is omitted, the value of the attribute is assumed to be equal to the current package name. Thus, it's redundant for `libjpeg` recipe to declare that it provides `libjpeg`, it's already implicitly assumed by Conan.

Other

dependencies

Conan recipes provide access to their dependencies via the `self.dependencies` attribute.

```
class Pkg(ConanFile):
    requires = "openssl/0.1"

    def generate(self):
        openssl = self.dependencies["openssl"]
        # access to members
        openssl.ref.version
        openssl.ref.revision # recipe revision
        openssl.options
        openssl.settings
```

See also:

Read here *the complete reference of self.dependencies*.

conf

In the `self.conf` attribute we can find all the conf entries declared in the <MISSING PAGE> [conf] section of the profiles. in addition of the declared <MISSING PAGE> self.conf_info entries from the first level tool requirements. The profile entries have priority.

```
from conan import ConanFile

class MyConsumer(ConanFile):
```

(continues on next page)

(continued from previous page)

```
tool_requires = "my_android_ndk/1.0"

def generate(self):
    # This is declared in the tool_requires
    self.output.info("NDK host: %s" % self.conf.get("tools.android:ndk_path"))
    # This is declared in the profile at [conf] section
    self.output.info("Custom var1: %s" % self.conf.get("user.custom.var1"))
```

revision_mode

This attribute allow each recipe to declare how the revision for the recipe itself should be computed. It can take two different values:

- `"hash"` (by default): Conan will use the checksum hash of the recipe manifest to compute the revision for the recipe.
- `"scm"`: the commit ID will be used as the recipe revision if it belongs to a known repository system (Git or SVN). If there is no repository it will raise an error.

upload_policy

Controls when the current package built binaries are uploaded or not

- `"skip"`: The precompiled binaries are not uploaded. This is useful for "installer" packages that just download and unzip something heavy (e.g. android-ndk), and is useful together with the `build_policy = "missing"`

  ```
  class Pkg(ConanFile):
      upload_policy = "skip"
  ```

required_conan_version

Recipes can define a module level `required_conan_version` that defines a valid version range of Conan versions that can load and understand the current `conanfile.py`. The syntax is:

```
from conan import ConanFile

required_conan_version = ">=2.0"

class Pkg(ConanFile):
    pass
```

Version ranges as in `requires` are allowed. Also there is a `global.conf` file `core:required_conan_version` configuration that can define a global minimum, maximum or exact Conan version to run, which can be very convenient to maintain teams of developers and CI machines to use the desired range of versions.

7.2.2 Methods

build()

The `build()` method is used to define the build from source of the package. In practice this means calling some build system, which could be done explicitly or using any of the build helpers provided by Conan:

```
from conan.tools.cmake import CMake

class Pkg(ConanFile):

    def build(self):
        # Either using some of the Conan built-in helpers
        cmake = CMake(self)
        cmake.configure()  # equivalent to self.run("cmake . <other args>")
        cmake.build() # equivalent to self.run("cmake --build . <other args>")
        cmake.test()  # equivalent to self.run("cmake --target=RUN_TESTS")

        # Or it could run your own build system or scripts
        self.run("mybuildsystem . --configure")
        self.run("mybuildsystem . --build")
```

For more information about the existing built-in build system integrations, visit *Recipe tools*.

The `build()` method should be as simple as possible, just wrapping the command line invocations that a developer would do in the simplest possible way. The `generate()` method is the one responsible for preparing the build, creating toolchain files, CMake presets, or any other files which are necessary so developers could easily call the build system by hand. This allows for much better integrations with IDEs and improves the developer experience. The result is that in practice the `build()` method should be relatively simple.

The `build()` method is the right place to build and run unit tests, before packaging, and raising errors if those tests fail, interrupting the process, and not even packaging the final binaries. The built-in helpers will skip the unit tests if the `tools.build:skip_test` configuration is defined. For custom integrations, it is expected that the method checks this `conf` value in order to skip building and running tests, which can be useful for some CI scenarios.

The `build()` method runs once per unique configuration, so if there are some source operations like applying patches that are done conditionally to different configurations, they could be also applied in the `build()` method, before the actual build. It is important to note that in this case the `no_copy_source` attribute cannot be set to `True`.

Note: Best practices

- The `build()` method should be as simple as possible, the heavy lifting of preparing the build should happen in the `generate()` method in order to achieve a good developer experience that can easily build locally with just `conan install .`, plus directly calling the build system or opening their IDE.

See also:

Follow the *tutorial about building packages* for more information about building from sources.

build_id()

The `build_id()` method allows to re-use the same build to create different binary packages in the cache, potentially saving build time as it can avoid some unnecessary re-builds. It is therefore an optimization method.

In the general case, there is one build folder for each binary package, with the exact same `package_id` of the package. However this behavior can be changed, there are a couple of scenarios that this might be useful:

- The package build scripts generate several different configurations at once (like both debug and release artifacts) in the same run, without the possibility of building each configuration separately.

- The package build scripts generate one binary configuration, but different artifacts that can be packaged separately. For example if there are some test executables, you might want to create two packages: one just containing the library for general usage, and another one also containing the tests (for compliance, later reproducibility, debugging, etc).

In the first case, we could for example write:

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

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

This recipe will generate a final different package with a different `package_id` for debug and release configurations. But as the `build_id()` will generate the same `build_id` for any `build_type`, then just one folder and one `build()` will be done, building both debug and release artifacts, and then the `package()` method will be called for each configuration, and it should package the artifacts conditionally to the `self.settings.build_type` value. Different builds will still be executed if using different compilers or architectures.

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 `info_build` data, and it still produces a different id 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 different `build_id` if the package is for Windows, thus running `build()` just once for all `build_type` values. The behavior in any other OS will be the standard one, as if the `build_id()` method was not defined, running one different `build()` for each `build_type`.

Note: Best practices

Conan strongly recommends to use one package binary with its own `package_id` for each different configuration. The goal of the `build_id()` method is to deal with legacy build scripts that cannot easily be changed to do the build of one configuration each time.

---

build_requirements()

The `build_requirements()` method is functionally equivalent to the `requirements()` one, it is executed just after it. It is not strictly necessary, in theory everything that is inside this method, could be done in the end of the `requirements()` one. Still, `build_requirements()` is good for having a dedicated place to define `tool_requires` and `test_requires`:

```
def build_requirements(self):
   self.tool_requires("cmake/3.23.5")
   self.test_requires("gtest/1.13.0")
```

For simple cases the attribute syntax can be enough, like `tool_requires = "cmake/3.23.5"` and `test_requires = "gtest/1.13.0"`. The method form can be necessary for conditional or parameterized requirements.

The `tool_requires` and `test_requires` methods are just a specialized instance of `requires` with some predefined trait values. See the *requires() reference* for more information about traits.

tool_requires

The `tool_requires` is equivalent to `requires()` with the following traits:

- `build=True`. This dependency is in the "build" context, being necessary at build time, but not at application runtime, and will receive the "build" profile and configuration.
- `visible=False`. The dependency to a tool requirement is not propagated downstream. For example, one package can call `tool_requires("cmake/3.23.5")`, but that doesn't mean that the consumer packages also use `cmake`, they could even use a different build system, or a different version, without causing conflicts.
- `run=True`. This dependency has some executables or runtime that needs to be ran at build time.
- `headers=False` A tool requirement does not have headers.
- `libs=False`: A tool requirement does not have libraries to be linked by the consumer (if it had libraries they would be in the "build" context and could be incompatible with the "host" context of the consumer package).

test_requires

The `test_requires` is equivalent to `requires()` with the following traits:

- `test=True`. This dependency is a "test" dependency, existing in the "host" context, but not aiming to be part of the final product.
- `visible=False`. The dependency to a test requirement is not propagated downstream. For example, one package can call `self.test_requires("gtest/1.13.0")`, but that doesn't mean that the consumer packages also use `gtest`, they could even use a different test framework, or the same `gtest` with a different version, without causing conflicts.

It is possible to further modify individual traits of `tool_requires()` and `test_requires()` if necessary, for example:

```
def build_requirements(self):
    self.tool_requires("cmake/3.23.5", options={"shared": False})
```

Note: Best practices

- `tool_requires` are exclusively for build time **tools**, not for libraries that would be included and linked into the consumer package. For libraries with some special characteristics, use a `requires()` with custom trait values.
- The `self.test_requires()` and `self.tool_requires()` methods should exclusively be used in the `build_requirements()` method, with the only possible exception being the `requirements()` method. Using them in any other method is forbidden. To access information about dependencies when necessary in some methods, the *self.dependencies* attribute should be used.

See also:

- Follow the *tutorial about consuming Conan packages as tools*.

- Read the *tutorial about creating tool_requires packages*.

compatibility()

Warning: This is a **preview** feature

The `compatibility()` method implements the same binary compatibility mechanism than the *compatibility plugin*, but at the recipe level. In general, the global compatibility plugin should be good for most cases, and only require the recipe method for exceptional cases.

This method can be used in a *conanfile.py* to define packages that are compatible between each other. If there are no binaries available for the requested settings and options, this mechanism will retrieve the compatible package's binaries if they exist. This method should return a list of compatible configurations.

For example, if we want that binaries built with gcc versions 4.8, 4.7 and 4.6 to be considered compatible with the ones compiled with 4.9 we could declare a `compatibility()` method like this:

```
def compatibility(self):
    if self.settings.compiler == "gcc" and self.settings.compiler.version == "4.9":
        return [{"settings": [("compiler.version", v)]}
                for v in ("4.8", "4.7", "4.6")]
```

The format of the list returned is as shown below:

```
[
    {
        "settings": [(<setting>, <value>), (<setting>, <value>), ...],
        "options": [(<option>, <value>), (<option>, <value>), ...]
    },
    {
        "settings": [(<setting>, <value>), (<setting>, <value>), ...],
        "options": [(<option>, <value>), (<option>, <value>), ...]
    },
    ...
]
```

configure()

The `configure()` method should be used for the configuration of settings and options in the recipe for later use in the different methods like `generate()`, `build()` or `package()`. This method executes while building the dependency graph and expanding the packages dependencies, which means that when this method executes the dependencies are still not there, they do not exist, and it is not possible to access `self.dependencies`.

For example, for a C (not C++) library, the `compiler.libcxx` and `compiler.cppstd` settings shouldn't even exist during the `build()`. It is not only that they are not part of the `package_id`, but they shouldn't be used in the build process at all. They will be defined in the profile, because other packages in the graph can be C++ packages and need them, but it is the responsibility of this recipe to remove them so they are not used in the recipe:

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

def configure(self):
    # Not all compilers have libcxx subsetting, so we use rm_safe
    # to avoid exceptions
```

(continues on next page)

(continued from previous page)

```
    self.settings.rm_safe("compiler.libcxx")
    self.settings.rm_safe("compiler.cppstd")

def package_id(self):
    # No need to delete those settings here, they were already deleted
    pass
```

Likewise, for a package containing a library, the `fPIC` option really only applies when the library is compiled as a static library, but otherwise, the `fPIC` option doesn't make sense, so it should be removed:

```
options = {"shared": [True, False], "fPIC": [True, False]}
default_options = {"shared": False, "fPIC": True}

def configure(self):
    if self.options.shared:
        # fPIC might have been removed in config_options(), so we use rm_safe
        self.options.rm_safe("fPIC")
```

Recipes can suggest values for their dependencies options as `default_options = {"*:shared": True}`, but it is not possible to do that conditionally. For this purpose, it is also possible to use the `configure()` method:

```
def configure(self):
    if something:
        self.options["*"].shared = True
```

Note: Best practices

- Recall that it is **not** possible to define `settings` or `conf` values in recipes, they are read only.
- The definition of `options` values is only a "suggestion", depending on the graph computation, priorities, etc., the final value of `options` can be different from the one set by the recipe.

See also:

- Follow the *tutorial about recipe configuration methods*.

config_options()

The `config_options()` method is used to configure or constraint the available options in a package, **before** they are given a value. A typical use case is to remove an option in a given platform. For example, the `fPIC` flag doesn't exist in Windows, so it should be removed in this method like so:

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

The `config_options()` method executes before the `configure()` method, and before the actual assignment of the `options` values, but after settings are already defined.

See also:

- Follow the *tutorial about recipe configuration methods*.

export()

Equivalent to the `exports` attribute, but in method form. This method will be called at `export` time, which happens in the `conan export` and `conan create` commands, and it is intended to allow copying files from the user folder to the Conan cache folders, thus making files becoming part of the recipe. These sources will be uploaded to the servers together with the recipe, but are typically not downloaded unless the package is being built from source.

The current working directory will be `self.recipe_folder`, and it can use the `self.export_folder` as the destination folder for using `copy()` or your custom copy.

```
from conan import ConanFile
from conan.tools.files import copy

class Pkg(ConanFile):
    def export(self):
        # This LICENSE file is intended to be the license of the current conanfile.py␣
↪recipe
        # and go with it. It is not intended to be the license of the final package␣
↪(for that
        # purpose export_sources() would be recommended)
        copy(self, "LICENSE.md", self.recipe_folder, self.export_folder)
```

There are 2 files that are always exported to the cache, without being explicitly defined in the recipe: the `conanfile.py` recipe, and the `conandata.yml` file if it exists. The `conandata.yml` file is automatically loaded whenever the `conanfile.py` is loaded, becoming the `self.conan_data` attribute, so it is a intrinsic part of the recipe, so it is part of the "exported" recipe files, not of the "exported" source files.

Note: Best practices

- The recipe files must be configuration independent. Those files are common for all configurations, thus it is not possible to do conditional `export()` to different settings, options, or platforms. Do not try to do any kind of conditional export. If necessary export all the files necessary for all configurations at once.
- The exported files must be small. Exporting big files with the recipe will make the resolution of dependencies much slower the resolution.
- Only files that are necessary for the evaluation of the `conanfile.py` recipe must be exported with this method. Files necessary for building from sources should be exported with the `exports_sources` attribute or the *export_source()* method.

export_sources()

Equivalent to the `exports_sources` attribute, but in method form. This method will be called at `export` time, which happens in `conan export` and `conan create` commands, and it is intended to allow copying files from the user folder to the Conan cache folders, those files becoming part of the recipe sources. These sources will be uploaded to the servers together with the recipe, but are typically not downloaded unless the package is being built from source.

The current working directory will be `self.recipe_folder`, and it can use the `self.export_sources_folder` as the destination folder for using `copy()` or your custom copy.

```
from conan import ConanFile
from conan.tools.files import copy

class Pkg(ConanFile):
```

(continues on next page)

(continued from previous page)

```
    def export_sources(self):
        # This LICENSE.md is a source file intended to be part of the final package
        # it is not the license of the current recipe
        copy(self, "LICENSE.md", self.recipe_folder, self.export_sources_folder)
```

The method might be able to read files in the recipe folder and do something with it:

```
import os
from conan import ConanFile
from conan.tools.files import load, save

class Pkg(ConanFile):

    def export_sources(self):
        content = load(self, os.path.join(self.recipe_folder, "data.txt"))
        save(self, os.path.join(self.export_sources_folder, "myfile.txt"), content)
```

The `export_conandata_patches()` is a high-level helper function that does the export of the patches defined in the `conandata.yml` file, which could be later applied with `apply_conandata_patches()` in the `source()` method.

```
from conan.tools.files import export_conandata_patches

class Pkg(ConanFile):

    def export_sources(self):
        export_conandata_patches(self)
```

Note: Best practices

The recipe sources must be configuration independent. Those sources are common for all configurations, thus it is not possible to do conditional `export_sources()` to different settings, options, or platforms. Do not try to do any kind of conditional export. If necessary export all the files necessary for all configurations at once.

generate()

This method will run after the computation and installation of the dependency graph. This means that it will run after a **conan install** command, or when a package is being built in the cache, it will be run before calling the `build()` method.

The purpose of `generate()` is to prepare the build, generating the necessary files. These files would typically be:

- Files containing information to locate the dependencies, as `xxxx-config.cmake` CMake config scripts, or `xxxx.props` Visual Studio property files.
- Environment activation scripts, like `conanbuild.bat` or `conanbuild.sh`, that define all the necessary environment variables necessary for the build.
- Toolchain files, like `conan_toolchain.cmake`, that contains a mapping between the current Conan settings and options, and the build system specific syntax. `CMakePresets.json` for CMake users using modern versions.
- General purpose build information, as a `conanbuild.conf` file that could contain information for some toolchains like autotools to be used in the `build()` method.

- Specific build system files, like `conanvcvars.bat`, that contains the necessary Visual Studio *vcvars.bat* call for certain build systems like Ninja when compiling with the Microsoft compiler.

The idea is that the `generate()` method implements all the necessary logic, making both the user manual builds after a **`conan install`** very straightforward, and also the `build()` method logic simpler. The build produced by a user in their local flow should result in exactly the same one as the build done in the cache with a `conan create` without effort.

Generation of files happens in the `generators_folder` as defined by the current layout.

In many cases, the `generate()` method might not be necessary, and declaring the `generators` attribute could be enough:

```
from conan import ConanFile

class Pkg(ConanFile):
    generators = "CMakeDeps", "CMakeToolchain"
```

But the `generate()` method can explicitly instantiate those generators, use them conditionally (like using one build system in Windows, and another build system integration in other platforms), customize them, or provide a complete custom generation.

```
from conan import ConanFile
from conan.tools.cmake import CMakeToolchain

class Pkg(ConanFile):

    def generate(self):
        tc = CMakeToolchain(self)
        # customize toolchain "tc"
        tc.generate()
        # Or provide your own custom logic
```

The current working directory for the `generate()` method will be the `self.generators_folder` defined in the current layout.

For custom integrations, putting code in a common `python_require` would be a good way to avoid repetition in multiple recipes:

```
from conan import ConanFile
from conan.tools.cmake import CMakeToolchain

class Pkg(ConanFile):

    python_requires = "mygenerator/1.0"

    def generate(self):
        mygen = self.python_requires["mygenerator"].module.MyGenerator(self)
        # customize mygen behavior, like mygen.something= True
        mygen.generate()
```

In case it is necessary to collect or copy some files from the dependencies, it is also possible to do it in the `generate()` method, accessing `self.dependencies`. Listing the different include directories, lib directories from a dependency "mydep" would be possible like this:

```
from conan import ConanFile

class Pkg(ConanFile):
```

(continues on next page)

(continued from previous page)

```
    def generate(self):
        info = self.dependencies["mydep"].cpp_info
        self.output.info("**includedirs:{}**".format(info.includedirs))
        self.output.info("**libdirs:{}**".format(info.libdirs))
        self.output.info("**libs:{}**".format(info.libs))
```

And copying the shared libraries in Windows and OSX to the current build folder, could be done like:

```
from conan import ConanFile

class Pkg(ConanFile):

    def generate(self):
        for dep in self.dependencies.values():
            copy(self, "*.dylib", dep.cpp_info.libdir, self.build_folder)
            copy(self, "*.dll", dep.cpp_info.libdir, self.build_folder)
```

Note: Best practices

- Accessing dependencies `self.dependencies["mydep"].package_folder` is possible, but it will be `None` when the dependency "mydep" is in "editable" mode. If you plan to use editable packages, make sure to always reference the `cpp_info.xxxdirs` instead.

See also:

- Follow the *tutorial about preparing build from source in recipes*.

self.dependencies

Conan recipes provide access to their dependencies via the `self.dependencies` attribute. This attribute is generally used by generators like `CMakeDeps` or `MSBuildDeps` to generate the necessary files for the build.

This section documents the `self.dependencies` attribute, as it might be used by users both directly in recipe or indirectly to create custom build integrations and generators.

Dependencies interface

It is possible to access each one of the individual dependencies of the current recipe, with the following syntax:

```
class Pkg(ConanFile):
    requires = "openssl/0.1"

    def generate(self):
        openssl = self.dependencies["openssl"]
        # access to members
        openssl.ref.version
        openssl.ref.revision # recipe revision
        openssl.options
        openssl.settings
```

Some **important** points:

- All the information is **read only**. Any attempt to modify dependencies information is an error and can raise at any time, even if it doesn't raise yet.
- It is not possible either to call any methods or any attempt to reuse code from the dependencies via this mechanism.
- This information does not exist in some recipe methods, only in those methods that evaluate after the full dependency graph has been computed. It will not exist in `configure()`, `config_options`, `export()`, `export_source()`, `set_name()`, `set_version()`, `requirements()`, `build_requirements()`, `system_requirements()`, `source()`, `init()`, `layout()`. Any attempt to use it in these methods can raise an error at any time.
- At the moment, this information should only be used in `generate()` and `validate()` methods. For any other use, please submit a Github issue.

Not all fields of the dependency conanfile are exposed, the current fields are:

- **package_folder**: The folder location of the dependency package binary
- **recipe_folder**: The folder containing the `conanfile.py` (and other exported files) of the dependency
- **ref**: An object that contains `name`, `version`, `user`, `channel` and `revision` (recipe revision)
- **pref**: An object that contains `ref`, `package_id` and `revision` (package revision)
- **buildenv_info**: `Environment` object with the information of the environment necessary to build
- **runenv_info**: `Environment` object with the information of the environment necessary to run the app
- **cpp_info**: includedirs, libdirs, etc for the dependency.
- **settings**: The actual settings values of this dependency
- **settings_build**: The actual build settings values of this dependency
- **options**: The actual options values of this dependency
- **context**: The context (build, host) of this dependency
- **conf_info**: Configuration information of this dependency, intended to be applied to consumers.
- **dependencies**: The transitive dependencies of this dependency
- **is_build_context**: Return `True` if `context == "build"`.
- **conan_data**: The `conan_data` attribute of the dependency that comes from its `conandata.yml` file
- **license**: The `license` attribute of the dependency
- **description**: The `description` attribute of the dependency
- **homepage**: The `homepage` attribute of the dependency
- **url**: The `url` attribute of the dependency

Iterating dependencies

It is possible to iterate in a dict-like fashion all dependencies of a recipe. Take into account that `self.dependencies` contains all the current dependencies, both direct and transitive. Every upstream dependency of the current one that has some effect on it, will have an entry in this `self.dependencies`.

Iterating the dependencies can be done as:

```
requires = "zlib/1.2.11", "poco/1.9.4"

def generate(self):
    for require, dependency in self.dependencies.items():
        self.output.info("Dependency is direct={}: {}".format(require.direct,␣
↪dependency.ref))
```

will output:

```
conanfile.py (hello/0.1): Dependency is direct=True: zlib/1.2.11
conanfile.py (hello/0.1): Dependency is direct=True: poco/1.9.4
conanfile.py (hello/0.1): Dependency is direct=False: pcre/8.44
conanfile.py (hello/0.1): Dependency is direct=False: expat/2.4.1
conanfile.py (hello/0.1): Dependency is direct=False: sqlite3/3.35.5
conanfile.py (hello/0.1): Dependency is direct=False: openssl/1.1.1k
conanfile.py (hello/0.1): Dependency is direct=False: bzip2/1.0.8
```

Where the `require` dictionary key is a "requirement", and can contain specifiers of the relation between the current recipe and the dependency. At the moment they can be:

- `require.direct`: boolean, `True` if it is direct dependency or `False` if it is a transitive one.
- `require.build`: boolean, `True` if it is a `build_require` in the build context, as `cmake`.
- `require.test`: boolean, `True` if its a `build_require` in the host context (defined with `self.test_requires()`), as `gtest`.

The `dependency` dictionary value is the read-only object described above that access the dependency attributes.

The `self.dependencies` contains some helpers to filter based on some criteria:

- `self.dependencies.host`: Will filter out requires with `build=True`, leaving regular dependencies like `zlib` or `poco`.
- `self.dependencies.direct_host`: Will filter out requires with `build=True` or `direct=False`
- `self.dependencies.build`: Will filter out requires with `build=False`, leaving only `tool_requires` in the build context, as `cmake`.
- `self.dependencies.direct_build`: Will filter out requires with `build=False` or `direct=False`
- `self.dependencies.test`: Will filter out requires with `build=True` or with `test=False`, leaving only test requirements as `gtest` in the host context.

They can be used in the same way:

```
requires = "zlib/1.2.11", "poco/1.9.4"

def generate(self):
    cmake = self.dependencies.direct_build["cmake"]
    for require, dependency in self.dependencies.build.items():
        # do something, only build deps here
```

Dependencies `cpp_info` interface

The `cpp_info` interface is heavily used by build systems to access the data. This object defines global and per-component attributes to access information like the include folders:

```python
def generate(self):
    cpp_info = self.dependencies["mydep"].cpp_info
    cpp_info.includedirs
    cpp_info.libdirs

    cpp_info.components["mycomp"].includedirs
    cpp_info.components["mycomp"].libdirs
```

All the paths declared in the `cppinfo` object (like `cpp_info.includedirs`) are absolute paths and works whether the dependency is in the cache or is an *editable package*.

See also:

Read more about the *CppInfo* model.

init()

This is an optional method for initializing conanfile values, designed for inheritance from `python_requires`. Assuming we have a `base/1.1` recipe:

Listing 12: base/conanfile.py

```python
from conan import ConanFile

class MyConanfileBase:
    license = "MyLicense"
    settings = "os", # tuple!


class PyReq(ConanFile):
    name = "base"
    version = "1.1"
```

We could reuse and inherit from it with:

Listing 13: pkg/conanfile.py

```python
from conan import ConanFile

class Pkg(ConanFile):
    license = "MIT"
    settings = "arch", # tuple!
    python_requires = "base/1.1"
    python_requires_extend = "base.MyConanfileBase"

    def init(self):
        base = self.python_requires["base"].module.MyConanfileBase
        self.settings = base.settings + self.settings  # Note, adding 2 tuples = tuple
        self.license = base.license  # License is overwritten
```

The final `Pkg` conanfile will have both `os` and `arch` as settings, and `MyLicense` as license.

This method can also be useful if you need to unconditionally initialize class attributes like `license` or `description` or any other from datafiles other than *conandata.yml*. For example, you can have a *json* file containing the information about the `license`, `description` and `author` for the library:

Listing 14: data.json

```
{"license": "MIT", "description": "This is my awesome library.", "author": "Me"}
```

Then, you can load that information from the `init()` method:

```
import os
import json
from conan import ConanFile
from conan.tools.files import load


class Pkg(ConanFile):
    exports = "data.json" # Important that it is exported with the recipe

    def init(self):
        data = load(self, os.path.join(self.recipe_folder, "data.json"))
        d = json.loads(data)
        self.license = d["license"]
        self.description = d["description"]
        self.author = d["author"]
```

Note: Best practices

- Try to keep your `python_requires` as simple as possible, and do not reuse attributes from them (the main need for the `init()` method), trying to avoid the complexity of this `init()` method. In general inheritance can have more issues than composition (or in other words "use composition over inheritance" as a general programming good practice), so try to avoid it if possible.
- Do not abuse `init()` for other purposes other than listed here, nor use the Python private `ConanFile.__init__` constructor.
- The `init()` method executes at recipe load time. It cannot contain conditionals on settings, options, conf, or use any dependencies information other than the above `python_requires`.

layout()

In the layout() method you can adjust `self.folders` and `self.cpp`.

self.folders

- **self.folders.source** (Defaulted to `""`): Specifies a subfolder where the sources are. The `self.source_folder` attribute inside the `source(self)` and `build(self)` methods will be set with this subfolder. The *current working directory* in the `source(self)` method will include this subfolder. The *export_sources* and *exports* sources will also be copied to the root source directory. It is used in the cache when running **conan create** (relative to the cache source folder) as well as in a local folder when running **conan build** (relative to the local current folder).
- **self.folders.build** (Defaulted to `""`): Specifies a subfolder where the files from the build are. The `self.build_folder` attribute and the *current working directory* inside the `build(self)` method will be set with this subfolder. It is used in the cache when running **conan create** (relative to the cache source folder) as well as in a local folder when running **conan build** (relative to the local current folder).

- **self.folders.generators** (Defaulted to `""`): Specifies a subfolder in which to write the files from the generators and the toolchains. In the cache, when running **conan create**, this subfolder will be relative to the root build folder and when running the **conan install** command it will be relative to the current working directory.
- **self.folders.root** (Defaulted to `None`): Specifies a parent directory where the sources, generators, etc., are located specifically when the `conanfile.py` is located in a separated subdirectory. Check *this example* on how to use **self.folders.root**.
- **self.folders.subproject** (Defaulted to `None`): Specifies a subfolder where the `conanfile.py` is relative to the project root. This is particularly useful for *layouts with multiple subprojects*
- **self.folders.build_folder_vars** (Defaulted to `None`): Use settings and options to produce a different build folder and different CMake presets names.

self.cpp

The `layout()` method allows to declare `cpp_info` objects not only for the final package (like the classic approach with the `self.cpp_info` in the `package_info(self)` method) but for the `self.source_folder` and `self.build_folder`.

The fields of the `cpp_info` objects at `self.cpp.build` and `self.cpp.source` are the same described *here*. Components are also supported.

Properties to declare all the information needed by the consumers of a package: include directories, library names, library paths... Used both for *editable packages* and regular packages in the cache.

There are four instances available, only while running the following methods:

- **At `layout(self)` method:**
  - **self.cpp.package**: For a regular package being used from the Conan cache.
  - **self.cpp.source**: For "editable" packages, to describe the artifacts under `self.source_folder`.
  - **self.cpp.build**: For "editable" packages, to describe the artifacts under `self.build_folder`.

```
def layout(self):
    ...
    self.folders.source = "src"
    self.folders.build = "build"

    # In the local folder (before a conan create) the artifacts can␣
↪be found:
    self.cpp.source.includedirs = ["my_includes"]
    self.cpp.build.libdirs = ["lib/x86_64"]
    self.cpp.build.libs = ["foo"]

    # In the Conan cache, we packaged everything at the default␣
↪standard directories, the library to link
    # is "foo"
    self.cpp.package.libs = ["foo"]
```

See also:

Read more about the usage of the `layout()` in *this tutorial* and Conan package layout *here*.

Environment variables and configuration

There are some packages that might define some environment variables in their `package_info()` method via `self.buildenv_info`, `self.runenv_info`. Other packages can also use `self.conf_info` to pass configuration to their consumers.

This is not an issue as long as the value of those environment variables or configuration do not require using the `self.package_folder`. If they do, then their values will not be correct for the "source" and "build" layouts. Something like this will be **broken** when used in `editable` mode:

```
import os
from conan import ConanFile

class SayConan(ConanFile):
    ...
    def package_info(self):
        # This is BROKEN if we put this package in editable mode
        self.runenv_info.define_path("MYDATA_PATH",
                                     os.path.join(self.package_folder, "my/data/path
↪"))
```

When the package is in editable mode, for example, `self.package_folder` is `None`, as obviously there is no package yet. The solution is to define it in the `layout()` method, in the same way the `cpp_info` can be defined there:

```
from conan import ConanFile

class SayConan(ConanFile):
    ...
    def layout(self):
        # The final path will be relative to the self.source_folder
        self.layouts.source.buildenv_info.define_path("MYDATA_PATH", "my/source/data/
↪path")
        # The final path will be relative to the self.build_folder
        self.layouts.build.buildenv_info.define_path("MYDATA_PATH2", "my/build/data/
↪path")
        # The final path will be relative to the self.build_folder
        self.layouts.build.conf_info.define_path("MYCONF", "my_conf_folder")
```

The `layouts` object contains `source`, `build` and `package` scopes, and each one contains one instance of `buildenv_info`, `runenv_info` and `conf_info`.

package()

The `package()` method is in charge of copying files from the `source_folder` and the temporary `build_folder` to the `package_folder`, copying only those files and artifacts that will be part of the final package, like headers, compiler static and shared libraries, executables, license files, etc.

The `package()` method will be called once per different configuration that is creating a new package binary, which happens with `conan install --build=pkg*`, `conan create` and `conan export-pkg` commands.

There are 2 main ways the `package()` method can do such a copy. The first one is an explicit `copy()` from the origin `source_folder` and `build_folder` to the `package folder`:

```
from conan import ConanFile
from conan.tools.files import copy
```

(continues on next page)

(continued from previous page)

```
class Pkg(ConanFile):

    def package(self):
        # copying headers from source_folder
        copy(self, "*.h", join(self.source_folder, "include"), join(self.package_
↪folder, "include"))
        # copying compiled .lib from build folder
        copy(self, "*.lib", self.build_folder, join(self.package_folder, "lib"), keep_
↪path=False)
```

The second way is to use the `install` functionality of some build systems, provided that the build scripts implement such functionality. For example if the `CMakeLists.txt` of a package implements the correct CMake `INSTALL` instructions, it is possible to do:

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

Also, it is possible to combine both approaches, doing `cmake.install()` and also adding some `copy()` calls, for example to make sure some "License.txt" file is packaged that was not taken into account by the CMakeLists.txt script.

It is also possible to use conditionals in the `package()` method, because different platforms might have different artifacts in different locations:

```
def package(self):
    if self.settings.os == "Windows":
        copy(self, "*.lib", src=os.path.join(self.build_folder, "libs"), ...)
        copy(self, "*.dll", ....)
    else:
        copy(self, "*.lib", src=os.path.join(self.build_folder, "build", "libs"), ...)
```

Though in most situations it might not be necessary, because pattern based copy will likely not find wrong artifacts like `*.dll` in a non-Windows build.

The `package()` method is also the one called when packaging precompiled binaries with `conan export-pkg`. In this case the `self.source_folder` and `self.build_folder` refer to user space folders, as defined by the `layout()` method and the only folder in the Conan cache will be `self.package_folder`.

Note: Best practices

The `cmake.install()` functionality should be called in the `package()` method, not in the `build()` method. It is not necessary to reuse the `CMake(self)` object, it shouldn't be reused among methods. Creating a new instance in every method is the recommended approach.

See also:

See :ref:‘ the package() method tutorial<creating_packages_package_method>‘ for more information.

package_id()

Conan computes a unique `package_id` reference for each configuration, including `settings`, `options` and `dependencies` versions. This `package_id()` method allows some customizations and changes over the computed `package_id`, in general with the goal to relax some of the global binary compatibility assumptions.

The general rule is that every different value of `settings` and `options` creates a different `package_id`. This rule can be relaxed or expanded following different approaches:

- A given package recipe can decide in its `package_id()` that the final binary is independent of some settings, for example if it is a header-only library, that uses input settings to build some tests, it might completely clear all configuration, so the resulting `package_id` is always the same irrespective of the inputs. Likewise a C library might want to remove the effect of `compiler.cppstd` and/or `compiler.libcxx` from its binary `package_id`, because as a C library, its binary will be independent.
- A given package recipe can implement some partial erasure of information, for example to obtain the same `package_id` for a range of compiler versions. This type of binary compatibility is in general better addressed with the global `compatibility` plugin, or with the `compatibility()` method if the global plugin is not enough.
- A package recipe can decide to inject extra variability in its computed `package_id`, adding `conf` items or "target" settings.

Information erasure

This is a `package_id` relaxing strategy. Let's check the first case: a header-only library, that has input `settings`, because it still wants to use them for some unit-tests in its `build()` method. In order to have exactly one final binary for all configurations, because the final artifact should be identical in all cases (just the header files), it would be necessary to do:

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

def build(self):
    cmake = CMake(self) # need specific settings to build
    ...
    cmake.test()  # running unit tests for the current configuration

def package_id(self):
    # Completely clear all the settings from the ``package_id`` information ("info"␣
↪object)
    # All resulting ``package_id`` will be the same, irrespective of configuration
    self.info.settings.clear()
```

Warning: The modifications of the information always happen over the `self.info` object, not on `self.settings` or `self.options`

If a package is just a C library, but it couldn't remove the `compiler.cppstd` and `compiler.libcxx` in the `configure()` method (the recommended approach for most cases, to guarantee those flags are not used in the build), because there are C++ unit tests to the C library, then as the tests are not packaged and the final binary will be independent of C++, those could be removed with:

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

def build(self):
    # building C++ tests for a C library

def package_id(self):
    del self.info.settings.compiler.cppstd
    # Some compilers might not declare libcxx subsetting
    self.info.settings.rm_safe("compiler.libcxx")
```

If a package is building an executable to be used as a tool, and only 1 executable for each OS and architecture is desired to be more efficient, the `package_id()` could remove the other settings and options if existing:

```
# this will be a "tool_require"
package_type = "application"
settings = "os", "compiler", "arch", "build_type"

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

Note that this doesn't mean that the `compiler` and `build_type` should be removed for every application executable. For other things that are not tools, but final products to release, the most common situation is that maintaining the different builds for the different compilers, compiler versions, build types, etc. is the best approach. It also means that we are erasing some information. We will not have the information of the compiler and build type that was used for the binary that we are using (it will not be in the `conan list` output, and it will not be in the server metadata either). If we compile a new binary with a different compiler or build type, it will create a new package revision under the same `package_id`.

Partial information erasure

It is also possible to partially erase information for given subsets of values. For example, if we want to have the same `package_id` for all the binaries compiled with `gcc` between versions 4.5 and 5.0, we can do:

```
def package_id(self):
    v = Version(str(self.settings.compiler.version))
    if self.settings.compiler == "gcc" and (v >= "4.5" and v < "5.0"):
        # The assigned string can be arbitrary
        self.info.settings.compiler.version = "GCC 4 between 4.5 and 5.0"
```

This will result in all other compilers rather than `gcc` and other versions outside of that range to have a different `package_id`, but there will be only 1 `package_id` binary for all `gcc 4.5-5.0` versions. This also has the disadvantage mentioned above about losing the information that created this binary.

This approach is not recommended in the general case, and it would be better approached with the global `compatibility` plugin or the recipe `compatibility()` method.

Adding information

There is some information not added by default to the `package_id`. If we are creating a package for a tool, to be used as a `tool_require`, and it happens that such package binary will be different for each "target" configuration, like it is the case for some cross-compilers, if the compiler itself might be different for the different architectures that it is targeting, it will be necessary to add the `settings_target` to the `package_id` with:

```
def package_id(self):
    self.info.settings_target = self.settings_target
```

The `conf` items do not affect the `package_id` by default. It is possible to explicitly make them part of it at the recipe level with:

```
def package_id(self):
    self.info.conf.define("user.myconf:myitem", self.conf.get("user.myconf:myitem"))
```

Although this can be achieved for all recipes without the `package_id()` method, using the `tools.info.package_id:confs = ["user.myconf:myitem"]` configuration.

See also:

See *the tutorial about header-only packages* for explanations about the `package_id()` method.

package_info()

The `package_info()` method is the one responsible of defining the information to the consumers of the package, so those consumers can easily and automatically consume this package. The `generate()` method of the consumers is the place where the information defined in the `package_info()` will be mapped to the specific build system of the consumer. Then, if we want a package to be consumed by different build systems (like it happens with ConanCenter recipes for the community), it is very important that this information is complete.

Important: This method defines information exclusively for **consumers** of this package, not for itself. This method executes after the binary has been built and packaged. The information that is consumed in the build should be processed in `generate()` method.

cpp_info: Library and build information

Each package has to specify certain build information for its consumers. This can be done in the `cpp_info` attribute.

```
# Binaries to link
self.cpp_info.libs = []  # The libs to link against
self.cpp_info.system_libs = []  # System libs to link against
self.cpp_info.frameworks = []  # OSX frameworks that consumers will link against
self.cpp_info.objects = []  # precompiled objects like .obj .o that consumers will␣
↪link
# Directories
self.cpp_info.includedirs = ['include']  # Ordered list of include paths
self.cpp_info.libdirs = ['lib']  # Directories where libraries can be found
self.cpp_info.bindirs = ['bin']  # Directories where executables and shared libs can␣
↪be found
self.cpp_info.resdirs = []  # Directories where resources, data, etc. can be found
self.cpp_info.srcdirs = []  # Directories where sources can be found (debugging,␣
↪reusing sources)
self.cpp_info.builddirs = []  # Directories where build scripts for consumers can be␣
↪found
self.cpp_info.frameworkdirs = []  # Directories where OSX frameworks can be found
# Flags
self.cpp_info.defines = []  # preprocessor definitions
self.cpp_info.cflags = []  # pure C flags
self.cpp_info.cxxflags = []  # C++ compilation flags
self.cpp_info.sharedlinkflags = []  # linker flags
self.cpp_info.exelinkflags = []  # linker flags
# Properties
self.cpp_info.set_property("property_name", "property_value")
# Structure
self.cpp_info.components # Dictionary-like structure to define the different␣
↪components a package may have
self.cpp_info.requires # List of components from requirements that need to be␣
↪propagated downstream
```

Binaries to link:

- **libs**: Ordered list of compiled libraries (contained in the package) the consumers should link. Empty by default.

- **system_libs**: Ordered list of system libs (not contained in the package) the consumers should link. Empty by default.
- **frameworks**: Ordered list of OSX frameworks (contained or not in the package), the consumers should link. Empty by default.
- **objects**: Ordered list of precompiled objects (.obj, .o) contained in the package the consumers should link. Empty by default

Directories:

- **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.
- **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.
- **bindirs**: List of relative paths (starting from the package root) of directories in which to find library runtime binaries (like executable Windows .dlls). By default it is initialized to `['bin']`, 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 empty.
- **srcdirs**: List of relative paths (starting from the package root) of directories in which to find sources (like .c, .cpp). By default it is empty. It might be used to store sources (for later debugging of packages, or to reuse those sources building them in other packages too).
- **builddirs**: List of relative paths (starting from package root) of directories that can contain build scripts that could be used by the consumers. Empty by default.
- **frameworkdirs**: List of relative paths (starting from the package root), of directories containing OSX frameworks.

Flags:

- **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**, **cxxflags**, **sharedlinkflags**, **exelinkflags**: List of flags that the consumer should activate for proper behavior. Rarely used.

Properties: - **set_property()** allows to define some built-in and user general properties to be propagated with the `cpp_info` model for consumers. They might contain build-system specific information. Some built-in properties are `cmake_file_name`, `cmake_target_name`, `pkg_config_name`, that can define specific behavior for `CMakeDeps` or `PkgConfigDeps` generators. For more information about these, read the specific build system integration documentation.

Structure:

- **components**: Dictionary with names as keys and a component object as value to model the different components a package may have: libraries, executables…
- **requires**: **Experimental** List of components from the requirements this package (and its consumers) should link with. It will be used by generators that add support for components features.

It is common that different configurations will produce different `package_info`, for example, the library names might change in different OSs, or different `system_libs` will be used depending on the compiler and OS:

```
settings = "os", "compiler", "arch", "build_type"
options = {"shared": [True, False]}

def package_info(self):
    if not self.settings.os == "Windows":
```

(continues on next page)

(continued from previous page)

```
        self.cpp_info.libs = ["zmq-static"] if not self.options.shared else ["zmq"]
    else:
        ...

    if not self.options.shared:
        self.cpp_info.defines = ["ZMQ_STATIC"]
    if self.settings.os == "Windows" and self.settings.compiler == "msvc":
        self.cpp_info.system_libs.append("ws2_32")
```

Properties

Any CppInfo object can declare "properties" that can be read by the generators. The value of a property can be of any type. Check each generator reference to see the properties used on it.

```
def set_property(self, property_name, value)
def get_property(self, property_name):
```

Example:

```
def package_info(self):
    self.cpp_info.set_property("cmake_find_mode", "both")
```

Components

If your package is composed by more than one library, it is possible to declare components that allow to define a `CppInfo` object per each of those libraries and also requirements between them and to components of other packages (the following case is not a real example):

```
def package_info(self):
    self.cpp_info.components["crypto"].set_property("cmake_file_name", "Crypto")
    self.cpp_info.components["crypto"].libs = ["libcrypto"]
    self.cpp_info.components["crypto"].defines = ["DEFINE_CRYPTO=1"]
    self.cpp_info.components["crypto"].requires = ["zlib::zlib"]  # Depends on all␣
↪components in zlib package

    self.cpp_info.components["ssl"].set_property("cmake_file_name", "SSL")
    self.cpp_info.components["ssl"].includedirs = ["include/headers_ssl"]
    self.cpp_info.components["ssl"].libs = ["libssl"]
    self.cpp_info.components["ssl"].requires = ["crypto",
                                                "boost::headers"]  # Depends on␣
↪headers component in boost package

    obj_ext = "obj" if platform.system() == "Windows" else "o"
    self.cpp_info.components["ssl-objs"].objects = [os.path.join("lib", "ssl-object.{}
↪".format(obj_ext))]
```

Dependencies among components and to components of other requirements can be defined using the `requires` attribute and the name of the component. The dependency graph for components will be calculated and values will be aggregated in the correct order for each field.

buildenv_info, runenv_info

The `buildenv_info` and `runenv_info` attributes are `Environment` objects that allow to define information for the consumers in the form of environment variables. They can use any of the `Environment` methods to define such information:

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

def package_info(self):
    self.buildenv_info.define("MYVAR", "1")
    self.buildenv_info.prepend_path("MYPATH", "my/path")
    if self.settings.os == "Android":
        arch = "myarmarch" if self.settings.arch=="armv8" else "otherarch"
        self.buildenv_info.append("MY_ANDROID_ARCH", f"android-{arch}")

    self.runenv_info.append_path("MYRUNPATH", "my/run/path")
    if self.settins.os == "Windows":
        self.runenv_info.define_path("MYPKGHOME", "my/home")
```

Note that these objects are not tied to either regular `requires` or `tool_requires`, any package recipe can use both. The difference between `buildenv_info` and `runenv_info` is that the former is applied when Conan is building something from source, like in the `build()` method, while the later would be used when executing something in the "host" context that would need the runtime activated.

Conan `VirtualBuildEnv` generator will be used by default in consumers, collecting the information from `buildenv_info` (and some `runenv_info` from the "build" context) to create the `conanbuild` environment script, which runs by default in all `self.run(cmd, env="conanbuild")` calls. The `VirtualRunEnv` generator will also be used by default in consumers collecting the `runenv_info` from the "host" context creating the `conanrun` environment script, which can be explicitly used with `self.run(<cmd>, env="conanrun")`.

Note: Best practices

It is not necessary to add `bindirs` to the `PATH` environment variable, this will be automatically done by the consumer `VirtualBuildEnv` and `VirtualRunEnv` generators. Likewise, it is not necessary to add `includedirs`, `libdirs` or any other dirs to environment variables, as this information will be typically managed by other generators.

conf_info

`tool_requires` packages in the "build" context can transmit some `conf` configuration to its immediate consumers, with the `conf_info` attribute. For example, one Conan package packaging the AndroidNDK could do:

```
def package_info(self):
    self.conf_info.define_path("tools.android:ndk_path", "path/to/ndk/in/package")
```

`conf_info` from packages can still be overwritten from profiles values, because user profiles will have higher priority.

`Conf.`**`define`**`(`*name*, *value*`)`

Define a value for the given configuration name.

Parameters

- **`name`** – Name of the configuration.
- **`value`** – Value of the configuration.

```
def package_info(self):
    # Setting values
    self.conf_info.define("tools.microsoft.msbuild:verbosity", "Diagnostic")
    self.conf_info.define("tools.system.package_manager:sudo", True)
    self.conf_info.define("tools.microsoft.msbuild:max_cpu_count", 2)
    self.conf_info.define("user.myconf.build:ldflags", ["--flag1", "--flag2"])
    self.conf_info.define("tools.microsoft.msbuildtoolchain:compile_options", {
↪"ExceptionHandling": "Async"})
```

`Conf.`**`append`**(*name*, *value*)

Append a value to the given configuration name.

Parameters

- **`name`** – Name of the configuration.
- **`value`** – Value to append.

```
def package_info(self):
    # Modifying configuration list-like values
    self.conf_info.append("user.myconf.build:ldflags", "--flag3")  # == ["--flag1
↪", "--flag2", "--flag3"]
```

`Conf.`**`prepend`**(*name*, *value*)

Prepend a value to the given configuration name.

Parameters

- **`name`** – Name of the configuration.
- **`value`** – Value to prepend.

```
def package_info(self):
    self.conf_info.prepend("user.myconf.build:ldflags", "--flag0")  # == ["--flag0
↪", "--flag1", "--flag2", "--flag3"]
```

`Conf.`**`update`**(*name*, *value*)

Update the value to the given configuration name.

Parameters

- **`name`** – Name of the configuration.
- **`value`** – Value of the configuration.

```
def package_info(self):
    # Modifying configuration dict-like values
    self.conf_info.update("tools.microsoft.msbuildtoolchain:compile_options", {
↪"ExpandAttributedSource": "false"})
```

`Conf.`**`remove`**(*name*, *value*)

Remove a value from the given configuration name.

Parameters

- **`name`** – Name of the configuration.
- **`value`** – Value to remove.

```
def package_info(self):
    # Remove
    self.conf_info.remove("user.myconf.build:ldflags", "--flag1")  # == ["--flag0
↪", "--flag2", "--flag3"]
```

`Conf.unset(name)`

Clears the variable, equivalent to a unset or set XXX=

Parameters **name** – Name of the configuration.

```
def package_info(self):
    # Unset any value
    self.conf_info.unset("tools.microsoft.msbuildtoolchain:compile_options")
```

It is possible to define configuration in packages that are `tool_requires`. For example, assuming there is a package that bundles the *AndroidNDK*, it could define the location of such NDK to the `tools.android:ndk_path` configuration as:

```
import os
from conan import ConanFile

class Pkg(ConanFile):
    name = "android_ndk"

    def package_info(self):
        self.conf_info.define("tools.android:ndk_path", os.path.join(self.package_
↪folder, "ndk"))
```

Note that this only propagates from the immediate, direct `tool_requires` of a recipe.

Note: Best practices

- The `package_info()` method is not strictly necessary if you have other means of propagating information for consumers. For example, if your package creates `xxx-config.cmake` files at build time, and they are put in the final package, it might not be necessary to define `package_info()` at all, and in the consumer side the `CMakeDeps` would not be necessary either, as `CMakeToolchain` is able to inject the paths to locate the `xxx-config.cmake` files inside the packages. This approach can be good for private usage of Conan, albeit some limitations of CMake, like not being able to manage multi-configuration projects (like Visual Studio switching Debug/Release in the IDE, that `CMakeDeps` can provide), limitations in some cross-build scenarios using packages that are both libraries and build tools (like `protobuf`, that also `CMakeDeps` can handle).
- Providing a `package_info()` is very necessary if consumers can use different build systems, like in ConanCenter. In this case, it is necessary a bit of repetition, and coding the `package_info()` might feel duplicating the package `xxx-config.cmake`, but automatically extracting the info from CMake is not feasible at this moment.
- If you plan to use editables or the local development flow, there's a need to check the `layout()` and define the information for `self.cpp.build` and `self.cpp.source`.
- It is not necessary to add `bindirs` to the `PATH` environment variable, this will be automatically done by the consumer `VirtualBuildEnv` and `VirtualRunEnv` generators.
- The **paths** defined in `package_info()` shouldn't be converted to any specific format (like the one required by Windows subsystems). Instead, it is the responsibility of the consumer to translate these paths to the adequate format.

See also:

See *the defining package information tutorial* for more information.

requirements()

Requirement traits

Traits are properties of a requires clause. They determine how various parts of a dependency are treated and propagated by Conan. Values for traits are usually computed by Conan based on the dependency's *package_type*, but can also be specified manually.

A good introduction to traits is provided in the Advanced Dependencies Model in Conan 2.0 presentation.

In the example below `headers` and `libs` are traits.

```
self.requires("math/1.0", headers=True, libs=True)
```

headers

Indicates that there are headers that are going to be `#included` from this package at compile time. The dependency will be in the host context.

libs

The dependency contains some library or artifact that will be used at link time of the consumer. This trait will typically be `True` for direct shared and static libraries, but could be false for indirect static libraries that are consumed via a shared library. The dependency will be in the host context.

build

This dependency is a build tool, an application or executable, like cmake, that is used exclusively at build time. It is not linked/embedded into binaries, and will be in the build context.

run

This dependency contains some executables, either apps or shared libraries that need to be available to execute (typically in the path, or other system env-vars). This trait can be `True` for `build=False`, in that case, the package will contain some executables that can run in the host system when installing it, typically like an end-user application. This trait can be `True` for `build=True`, the package will contain executables that will run in the build context, typically while being used to build other packages.

visible

This `require` will be propagated downstream, even if it doesn't propagate `headers`, `libs` or `run` traits. Requirements that propagate downstream can cause version conflicts. This is typically `True`, because in most cases, having 2 different versions of the same library in the same dependency graph is at least complicated, if not directly violating ODR or causing linking errors. It can be set to `False` in advanced scenarios, when we want to use different versions of the same package during the build.

transitive_headers

If `True` the headers of the dependency will be visible downstream.

transitive_libs

If `True` the libraries to link with of the dependency will be visible downstream.

test

This requirement is a test library or framework, like Catch2 or gtest. It is mostly a library that needs to be included and linked, but that will not be propagated downstream.

package_id_mode

If the recipe wants to specify how the dependency version affects the current package `package_id`, can be directly specified here.

While it could be also done in the `package_id()` method, it seems simpler to be able to specify it in the `requires` while avoiding some ambiguities.

```
# We set the package_id_mode so it is part of the package_id
self.tool_requires("tool/1.1.1", package_id_mode="minor_mode")
```

Which would be equivalent to:

```
def package_id(self):
  self.info.requires["tool"].minor_mode()
```

force

This `requires` will force its version in the dependency graph upstream, overriding other existing versions even of transitive dependencies, and also solving potential existing conflicts. The downstream consunmers `force` trait always have higher priority.

override

The same as the `force` trait, but not adding a `direct` dependency. If there is no transitive dependency to override, this `require` will be discarded. This trait only exists at the time of defining a `requires`, but it will not exist as an actual `requires` once the graph is fully evaluated

direct

If the dependency is a direct one, that is, it has explicitly been declared by the current recipe, or if it is a transitive one.

set_name()

Dynamically define `name` attribute. This method would be rarely needed, as the only use case that makes sense is when a recipe is shared and used to create different packages with the same recipe. In most cases the recommended approach is to define the `name = "mypkg"` attribute in the recipe.

This method is executed only when the recipe is exported to the cache `conan create` and `conan export`, and when the recipe is being locally used, like with `conan install .`. In all other cases, the name of the package is fully defined, and `set_name()` will not be called, so do not rely on it for any other functionality different than defining the `self.name` value.

If the current package name was defined in a *name.txt* file, it would be possible to do:

```
from conan import ConanFile
from conan.tools.files import load

class Pkg(ConanFile):
    def set_name(self):
        # This will execute relatively to the current user directory (name.txt in cwd)
        self.name = load(self, "name.txt")
        # if "name.txt" is located relative to the conanfile.py better do:
        self.name = load(self, os.path.join(self.recipe_folder, "name.txt"))
```

The package name can also be defined in command line for some commands with `--name=xxxx` argument. If we want to prioritize the command line argument we should do:

```
from conan import ConanFile
from conan.tools.files import load

class Pkg(ConanFile):
    def set_name(self):
        # Command line ``--name=xxxx`` will be assigned first to self.name and have␣
↪priority
        self.name = self.name or load(self, "name.txt")
```

Warning: The `set_name()` method is an alternative to the `name` attribute. It is not advised or supported to define both a `name` attribute and a `set_name()` method.

set_version()

Dynamically define `version` attribute. This method might be needed when the same recipe is being used to create different versions of the same package, and such version is defined elsewhere, like in the git branch or in a text or build script file. This would be a common situation.

This method is executed only when the recipe is exported to the cache `conan create` and `conan export`, and when the recipe is being locally used, like with `conan install .`. In all other cases, the version of the package is fully defined, and `set_version()` will not be called, so do not rely on it for any other functionality different than defining the `self.version` value.

If the current package version was defined in a *version.txt* file, it would be possible to do:

```
from conan import ConanFile
from conan.tools.files import load

class Pkg(ConanFile):
```

(continues on next page)

(continued from previous page)

```
    def set_version(self):
        # This will execute relatively to the current user directory (version.txt in␣
↪cwd)
        self.version = load(self, "version.txt")
        # if "version.txt" is located relative to the conanfile.py better do:
        self.version = load(self, os.path.join(self.recipe_folder, "version.txt"))
```

The package version can also be defined in command line for some commands with `--version=xxxx` argument. If we want to prioritize the command line argument we should do:

```
from conan import ConanFile
from conan.tools.files import load

class Pkg(ConanFile):
    def set_version(self):
        # Command line ``--version=xxxx`` will be assigned first to self.version and␣
↪have priority
        self.version = self.version or load(self, "version.txt")
```

A common use case could be to define the `version` dynamically from some version control mechanism, like the current git tag. This could be done with:

```
from conan import ConanFile
from conan.tools.scm import Git

class Pkg(ConanFile):
    name = "pkg"

    def set_version(self):
        git = Git(self, self.recipe_folder)
        self.version = git.run("describe --tags")
```

Warning: The `set_version()` method is an alternative to the `version` attribute. It is not advised or supported to define both a `version` attribute and a `set_version()` method.

source()

The `source()` method can be used to retrieve the necessary source code to build a package from source, and to apply patches to such source code if necessary. It will be called when a package is being built from source, like with `conan create` or `conan install --build=pkg*`, but it will not be called if a package pre-compiled binary is being used. That means that the source code will not be downloaded if a pre-compiled binary exists.

The `source()` method can implement different strategies for retrieving the source code:

- Fetching the source code for a third party library:
  - Using a `Git(self).clone()` to clone a Git repository
  - Executing a `download()` + `unzip()` or a combined `get()` (internally does download + unzip) to download a tarball, tgz, or zip archive.
- Fetching the source code for itself, from its repository, whose coordinates have been captured in the `conandata.yml` file in the `export()` method. This is the strategy that would be used to manage the source code for packages in which the `conanfile.py` lives in the package itself, but that for some reason we

don't want to put the source code in the recipe (like not distributing our source code, but being able to distribute our package binaries).

The `source()` method executes in the `self.source_folder`, the current working directory will be equal to that folder (which value is derived from `layout()` method).

A `source()` implementation might use the convenient `get()` helper, or use its own mechanisms or other Conan helpers for the task, something like:

```
import os
import shutil

from conan import ConanFile
from conan.tools.files import download, unzip, check_sha1


class PocoConan(ConanFile):
    name = "poco"
    version = "1.6.0"

    def source(self):
        zip_name = f"poco-{self.version}-release.zip"
        # Immutable source .zip
        download(self, f"https://github.com/pocoproject/poco/archive/poco-{self.
↪version}-release.zip", zip_name)
        # Recommended practice, always check hashes of downloaded files
        check_sha1(self, zip_name, "8d87812ce591ced8ce3a022beec1df1c8b2fac87")
        unzip(self, zip_name)
        shutil.move(f"poco-poco-{self.version}-release", "poco")
        os.unlink(zip_name)
```

Applying patches to downloaded sources can be done (and should be done) in the `source()` method if those patches apply to all possible configurations. As explained below, it is not possible to introduce conditionals in the `source()` method. If the patches are in file form, those patches must be exported together with the recipe, so they can be used whenever a build from source is fired.

It is possible to apply patches with:

- Your own or `git` patches utilities
- The Conan built-in `patch()` utility to explicitly apply patches one by one
- Apply the `apply_conandata_patches()` Conan utility to automatically apply all patches defined in `conandata.yml` file following some conventions.

Source caching

Once the `source()` method has been called, its result will be cached and reused for any build from source, for any configuration. That means that the retrieval of sources from the `source()` method should be completely independent of the configuration. It is not possible to implement conditionals on the `settings`, and in general, any attempt to apply any conditional logic to the `source()` method is wrong.

```
def source(self):
    if self.settings.compiler == "gcc":  # ERROR, will raise
        # download some source
```

Trying to bypass the Conan exception by using some other mechanism like:

```
def source(self):
    # Might work, but NOT recommended, try to avoid as much as possible
    if platform.system() == "Windows":
        # download something
    else:
        # download something different
```

Might apparently work if not doing any cross-build, and not recollecting sources in a different OS, but could be problematic otherwise.

To be completely safe, if different source code is necessary for different configurations, the recommended approach would be to retrieve that code conditionally in the `build()` method.

Forced retrieval of sources

When working with a recipe in a user folder, it is easy to call the `source()` method and force the retrieval of the source code, that will be done in the same user folder, according to the `layout()` definition:

```
$ conan source .
```

Calling the `source()` method and forcing the retrieval of source code in the cache, for all or some dependencies, even if they are not being built from sources, is possible with the `tools.build:download_source=True` configuration. For example:

```
$ conan graph info . -c tools.build:download_source=True
```

Will compute the dependency graph, then call the `source()` method for all "host" packages in the graph (as the configuration by default is a "host" configuration, if you want also the sources for the "build" context `tool_requires`, you could use `-c:b tools.build:download_source=True`). It is possible to collect all the source folders from the json formatted output, or to automate recollection of all sources, a `deployer` could be used.

Likewise, it is possible to retrieve the sources for packages in other `create` and `install` commands, just by passing the configuration. Finally, as also configuration can be defined per-package, using `-c mypkg*:tools.build:download_source=True` would only retrieve the sources of packages matching the `mypkg*` pattern.

Note: Best practices

- The `source()` method should be the same for all configurations, it cannot be conditional to any configuration.
- The `source()` method should retrieve immutable sources. Using some branch name, HEAD, or a tarball whose URL is not immutable and is being overwritten is a bad practice and will lead to broken packages. Using a Git commit, a frozen Git release tag, or a fixed and versioned release tarballs is the expected input.
- Applying patches should be done by default in the `source()` method, except if the patches are exclusive for one configuration, in that case they could be applied in `build()` method.
- The `source()` method should not access nor manipulate files in other folders different to the `self.source_folder`. All the "exported" files are copied to the `self.source_folder` before calling it.

See also:

See *the tutorial about managing recipe sources* for more information.

system_requirements()

The `system_requirements()` method can be used to call the system package managers to install packages at the system level. In general, this should be reduced to a minimum, system packages are not modeleded dependencies, but it can be sometimes convenient to automate the installation of some system packages that are necessary for some Conan packages. For example, when creating a recipe to package the `opencv` library, we could realize that it needs in Linux the `gtk` libraries, but it might be undesired to create a package for them, because we want to make sure we use the system ones. We code

```
from conan import ConanFile
from conan.tools.system.package_manager import Apt

class OpenCV(ConanFile):
    name = "opencv"
    version = "4.0"

    def system_requirements(self):
        apt = Apt(self)
        apt.install(["libgtk-3-dev"], update=True, check=True)
```

For full reference of the built-in helpers for different system package managers read the *tools.system.package_manager documentation*.

Collecting system requirements

When `system_requirements()` uses some built-in `package_manager` helpers, it is possible to collect information about the installed or required system requirements. If we have the following `conanfile.py`:

```
from conan import ConanFile
from conan.tools.system.package_manager import Apt

class MyPkg(ConanFile):
    settings = "arch"

    def system_requirements(self):
        apt = Apt(self)
        apt.install(["pkg1", "pkg2"])
```

It is possible to display the installed system packages (with the default `tools.system.package_manager:mode` requirements will be checked, but not installed) with:

```
 # Assuming apt is the default or using explicitly
 #   -c tools.system.package_manager:tool=apt-get
 $ conan install . --format=json
"graph": {
     "nodes": [
         {
             "ref": "",
             "id": 0,
             "settings": {
                 "arch": "x86_64"
             },
             "system_requires": {
                 "apt-get": {
                     "install": [
                         "pkg1",
```

(continues on next page)

(continued from previous page)

```
                    "pkg2"
                ],
                "missing": []
            }
        },
```

A similar result can be obtained without even installing binaries, we could use the `report` or `report-installed` modes. The `report` mode displays the `install` packages, those are the packages that are required to be installed, irrespective of whether they are actually installed or not. The `report` mode does not check the system for those package, so it could even be ran in another OS:

```
$ conan graph info . -c tools.system.package_manager:mode=report --format=json
...
"system_requires": {
    "apt-get": {
        "install": [
            "pkg1",
            "pkg2"
        ]
    }
}
```

On the other hand, the `report-installed` mode will do a check if the package is installed in the system or not, but not failing nor raising any error if it is not found:

```
$ conan graph info . -c tools.system.package_manager:mode=report-installed --
↪format=json
...
"system_requires": {
    "apt-get": {
        "install": [
            "pkg1",
            "pkg2"
        ],
        "missing": [
            "pkg1",
            "pkg2"
        ]
    }
}
```

test()

The `test()` method is only used for **test_package/conanfile.py**. It will execute immediately after `build()` has been called, and its goal is to run some executable or tests on binaries to prove the package is correctly created. Note that it is intended to be used as a test of the package: the headers are found, the libraries are found, it is possible to link, etc. But it is **not intended** to run unit, integration or functional tests.

It usually takes the form:

```
def test(self):
    if can_run(self):
        cmd = os.path.join(self.cpp.build.bindir, "example")
        self.run(cmd, env="conanrun")
```

See also:

See :ref:‘ the “testing packages” tutorial<tutorial_creating_test>‘ for more information.

validate()

The `validate()` method can be used to mark a package binary as “invalid”, or not working for the current configuration. For example, if we have a header-only library that doesn’t work in Windows, we could have the following `conanfile.py`:

```python
from conan import ConanFile
from conan.errors import ConanInvalidConfiguration

class Pkg(ConanFile):
    name = "pkg"
    version = "1.0"
    package_type = "header-library"
    settings = "os"

    def validate(self):
        if self.settings.os == "Windows":
            raise ConanInvalidConfiguration("Windows not supported")

    def package_id(self):
        self.info.clear()  # header-only
```

If we try to create this package in Windows, it will fail, but if we do it in Linux, it will succeed:

```
$ conan create . -s os=Windows # FAILS
...
ERROR: There are invalid packages:
pkg/1.0: Invalid: Windows not supported
$ conan create . -s os=Linux # WORKS
```

And if we try to use it in Windows, it will fail again:

```
$ conan install --requires=pkg/1.0 -s os=Windows # FAILS
...
ERROR: There are invalid packages:
pkg/1.0: Invalid: Windows not supported
```

When the `ConanInvalidConfiguration` causes an error, Conan application exit code will be `6`

It is possible to check the validity of a given graph without raising errors with the `conan graph info` command:

```
$ conan graph info --requires=pkg/1.0 -s os=Windows --filter=binary
conanfile:
ref: conanfile
binary: None
pkg/1.0#cfc18fcc7a50ead278a7c1820be74e56:
ref: pkg/1.0#cfc18fcc7a50ead278a7c1820be74e56
binary: Invalid
```

The `validate()` method is evaluated after the whole graph has been computed. This means that it can use the `self.dependencies` information to raise errors:

```
from conan import ConanFile
from conan.errors import ConanInvalidConfiguration

class Pkg(ConanFile):
    requires = "dep/0.1"

    def validate(self):
        if self.dependencies["dep"].options.myoption == 2:
            raise ConanInvalidConfiguration("Option 2 of 'dep' not supported")
```

Note: Best practices

The `configure()` method evaluates before the graph is complete, so it doesn't have the real values of the dependencies `options`. The `validate()` method is the one that should be checking those dependencies options values if necessary, not `configure()`.

See also:

- Follow the *tutorial about preparing build from source in recipes*.

validate_build()

The `validate_build()` method is used to verify if a package binary can be **built** with the current configuration. It is different than the `validate()` method which raises when the package cannot be **used** with the current configuration.

The `validate_build()` method can check the `self.settings` and `self.options` values to raise `ConanInvalidaConfiguration` if necessary.

```
from conan import ConanFile
from conan.errors import ConanInvalidConfiguration

class Pkg(ConanFile):
    name = "pkg"
    version = "1.0"
    settings = "os", "arch", "compiler", "build_type"

    def package_id(self):
        # For this package, it doesn't matter the compiler used for the binary package
        del self.info.settings.compiler

    def validate_build(self):
        # But we know this cannot be build with "gcc"
        if self.settings.compiler == "gcc":
            raise ConanInvalidConfiguration("This doesn't build in GCC")
```

This package cannot be created with the `gcc` compiler, but it can be created with other:

```
$ conan create . -s compiler=gcc
...
ERROR: There are invalid packages:
pkg/1.0: Cannot build for this configuration: This doesn't build in GCC

$ conan create . -s compiler=clang  # WORKS!
```

Once the package has been built, it can be consumed with that compiler:

```
$ conan install --requires=pkg/1.0 -s compiler=gcc # WORKS!
```

- *build()*: Contains the build instructions to build a package from source
- *build_id()*: Allows reusing the same build to create different package binaries
- *build_requirements()*: Defines `tool_requires` and `test_requires`
- *compatibility()*: Defines binary compatibility at the recipe level
- *configure()*: Allows configuring settings and options while computing dependencies
- *config_options()*: Configure options while computing dependency graph
- *export()*: Copies files that are part of the recipe
- *export_sources()*: Copies files that are part of the recipe sources
- *generate()*: Generates the files that are necessary for building the package
- *init()*: Special initialization of recipe when extending from `python_requires`
- *layout()*: Defines the relative project layout, source folders, build folders, etc.
- *package()*: Copies files from build folder to the package folder.
- *package_id()*: Defines special logic for computing the binary `package_id` identifier
- *package_info()*: Provide information for consumers of this package about libraries, folders, etc.
- *requirements()*: Define the dependencies of the package
- *set_name()*: Dynamically define the name of a package
- *set_version()*: Dynamically define the version of a package.
- *source()*: Define the dependencies of the package
- *system_requirements()*: Call system package managers like Apt to install system packages
- *test()*: Run some simple package test (exclusive of `test_package`)
- *validate()*: Define if the current package is invalid (cannot work) with the current configuration.
- *validate_build()*: Define if the current package cannot be created with the current configuration.

7.3 conanfile.txt

The `conanfile.txt` file is a simplified version of `conanfile.py`, aimed at simple consumption of dependencies, but it cannot be used to create a package. Also, it is not necessary to have a `conanfile.txt` for consuming dependencies, a `conanfile.py` is perfectly suited for simple consumption of dependencies.

It also provides a simplified functionality, for example it is not possible to express conditional requirements in `conanfile.txt`, and it will be necessary to use a `conanfile.py` for that. Read *Understanding the flexibility of using conanfile.py vs conanfile.txt* for more information about this.

7.3.1 [requires]

List of requirements, specifying the full reference. Equivalent to `self.requires(<ref>)` in `conanfile.py`.

```
[requires]
poco/1.9.4
zlib/1.2.11
```

This section supports references with version-ranges too:

```
[requires]
poco/[>1.0,<1.9]
zlib/1.2.11
```

And specific recipe revisions can be pinned too:

```
[requires]
zlib/1.2.13#revision1
boost/1.70.0#revision2
```

7.3.2 [tool_requires]

List of tool requirements (executable tools) specifying the full reference. Equivalent to `self.tool_requires()` in `conanfile.py`.

```
[tool_requires]
7zip/16.00
cmake/3.23.0
```

This section also supports version ranges and pinned recipe revisions, as above.

In practice the `[tool_requires]` will be always installed (same as `[requires]`) as installing from a *conanfile.txt* means that something is going to be built, so the tool requirements are indeed needed. Note however, that by default `tool_requires` live in the "build" context, they cannot be libraries to built with, just executable tools, and for example, using the `CMakeDeps` generator, they will not create CMake config files for them (an exception is possible, but it requires using a `conanfile.py`, read the *CMakeDeps reference* for more information).

7.3.3 [test_requires]

List of test requirements specifying the full reference. Equivalent to `self.test_requires()` in `conanfile.py`.

```
[tool_requires]
gtest/1.12.1
```

This section also supports version ranges and pinned recipe revisions, as above. The behavior of `test_requires` is totally equivalent to the `[requires]` section above, as the only difference is that `test_requires` are not propagated to consumers, but as a `conanfile.txt` is never creating a package that can be consumed, it is irrelevant. It is provided to maintain the equivalence with `conanfile.py`

7.3.4 [generators]

List of built-in generators to be used, equivalent to the `conanfile.py` `generators = "CMakeDeps", ...` attribute.

```
[requires]
poco/1.9.4
zlib/1.2.13

[generators]
CMakeDeps
CMakeToolchain
```

7.3.5 [options]

List of options scoped for each package with a pattern like **package_name*:option = Value**.

```
[requires]
poco/1.9.4
zlib/1.2.11

[generators]
CMakeDeps
CMakeToolchain

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

For example using `*:shared=True` will define `shared=True` for all packages in the dependency graph that have this option defined.

7.3.6 [layout]

You can specify one name of a predefined layout. The available values are:

- cmake_layout
- vs_layout
- bazel_layout (experimental)

```
[layout]
cmake_layout
```

7.3.7 Read more

Read *Understanding the flexibility of using conanfile.py vs conanfile.txt* for more information about conanfile.txt vs conanfile.py.

7.4 Recipe tools

Tools are all things that can be imported and used in Conan recipes.

The import path is always like:

```
from conan.tools.cmake import CMakeToolchain, CMakeDeps, CMake
from conan.tools.microsoft import MSBuildToolchain, MSBuildDeps, MSBuild
```

The main guidelines are:

- Everything that recipes can import belong to `from conan.tools`. Any other thing is private implementation and shouldn't be used in recipes.
- Only documented, public (not preceded by _) tools can be used in recipes.

Contents:

7.4.1 conan.tools.cmake

CMakeDeps

The `CMakeDeps` generator produces the necessary files for each dependency to be able to use the cmake `find_package()` function to locate the dependencies. It can be used like:

```
from conan import ConanFile

class App(ConanFile):
    settings = "os", "arch", "compiler", "build_type"
    requires = "hello/0.1"
    generators = "CMakeDeps"
```

The full instantiation, that allows custom configuration can be done in the `generate()` method:

```
from conan import ConanFile
from conan.tools.cmake import CMakeDeps

class App(ConanFile):
    settings = "os", "arch", "compiler", "build_type"
    requires = "hello/0.1"

    def generate(self):
        cmake = CMakeDeps(self)
        cmake.generate()
```

Listing 15: **CMakeLists.txt**

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

find_package(hello REQUIRED)

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

By default, for a `hello` requires, you need to use `find_package(hello)` and link with the target `hello::hello`. Check *the properties affecting CMakeDeps* like `cmake_target_name` to customize the file and the target names in the conanfile.py of the dependencies and their components.

Note: The `CMakeDeps` is intended to run with the `CMakeToolchain` generator. It will set `CMAKE_PREFIX_PATH` and `CMAKE_MODULE_PATH` to the right folder (`conanfile.generators_folder`) so CMake can locate the generated config/module files.

Generated files

- **XXX-config.cmake**: By default, the CMakeDeps generator will create config files declaring the targets for the dependencies and their components (if declared).
- **FindXXX.cmake**: Only when the property `cmake_find_mode` is set by the dependency with "module" or "both". See *The properties affecting CMakeDeps* is set in the dependency.
- **Other necessary *.cmake**: files like version, flags and directory data or configuration.

Customization

There are some attributes you can adjust in the created `CMakeDeps` object to change the default behavior:

configuration

Allows to define custom user CMake configuration besides the standard Release, Debug, etc ones.

```
def generate(self):
    deps = CMakeDeps(self)
    # By default, ``deps.configuration`` will be ``self.settings.build_type``
    if self.options["hello"].shared:
        # Assuming the current project ``CMakeLists.txt`` defines the ReleasedShared␣
↪configuration.
        deps.configuration = "ReleaseShared"
    deps.generate()
```

The `CMakeDeps` is a *multi-configuration* generator, it can correctly create files for Release/Debug configurations to be simultaneously used by IDEs like Visual Studio. In single configuration environments, it is necessary to have a configuration defined, which must be provided via the `cmake ... -DCMAKE_BUILD_TYPE=<build-type>` argument in command line (Conan will do it automatically when necessary, in the `CMake.configure()` helper).

build_context_activated

When you have a **build-require**, by default, the config files (*xxx-config.cmake*) files are not generated. But you can activate it using the **build_context_activated** attribute:

```
tool_requires = ["my_tool/0.0.1"]

def generate(self):
    cmake = CMakeDeps(self)
    # generate the config files for the tool require
    cmake.build_context_activated = ["my_tool"]
    cmake.generate()
```

build_context_suffix

When you have the same package as a **build-require** and as a **regular require** it will cause a conflict in the generator because the file names of the config files will collide as well as the targets names, variables names etc.

For example, this is a typical situation with some requirements (capnproto, protobuf...) that contain a tool used to generate source code at build time (so it is a **build_require**), but also providing a library to link to the final application, so you also have a **regular require**. Solving this conflict is specially important when we are cross-building because the tool (that will run in the building machine) belongs to a different binary package than the library, that will "run" in the host machine.

You can use the **build_context_suffix** attribute to specify a suffix for a requirement, so the files/targets/variables of the requirement in the build context (tool require) will be renamed:

```
tool_requires = ["my_tool/0.0.1"]
requires = ["my_tool/0.0.1"]

def generate(self):
    cmake = CMakeDeps(self)
    # generate the config files for the tool require
    cmake.build_context_activated = ["my_tool"]
    # disambiguate the files, targets, etc
    cmake.build_context_suffix = {"my_tool": "_BUILD"}
    cmake.generate()
```

build_context_build_modules

Also there is another issue with the **build_modules**. As you may know, the recipes of the requirements can declare a *cppinfo.build_modules* entry containing one or more **.cmake** files. When the requirement is found by the cmake `find_package()` function, Conan will include automatically these files.

By default, Conan will include only the build modules from the `host` context (regular requires) to avoid the collision, but you can change the default behavior.

Use the **build_context_build_modules** attribute to specify require names to include the **build_modules** from **tool_requires**:

```
tool_requires = ["my_tool/0.0.1"]

def generate(self):
    cmake = CMakeDeps(self)
    # generate the config files for the tool require
    cmake.build_context_activated = ["my_tool"]
    # Choose the build modules from "build" context
    cmake.build_context_build_modules = ["my_tool"]
    cmake.generate()
```

Reference

class CMakeDeps(*conanfile*)

generate()
This method will save the generated files to the conanfile.generators_folder

set_property(*dep*, *prop*, *value*, *build_context=False*)
Using this method you can overwrite the *property* values set by the Conan recipes from the consumer. This can be done for *cmake_file_name*, *cmake_target_name*, *cmake_find_mode*, *cmake_module_file_name* and *cmake_module_target_name* properties.

Parameters

- **dep** – Name of the dependency to set the *property*. For components use the syntax: `dep_name::component_name`.
- **prop** – Name of the *property*.
- **value** – Value of the property. Use `None` to invalidate any value set by the upstream recipe.
- **build_context** – Set to `True` if you want to set the property for a dependency that belongs to the build context (`False` by default).

get_cmake_package_name(*dep*, *module_mode=None*)
Get the name of the file for the find_package(XXX)

get_find_mode(*dep*)

Parameters **dep** – requirement

Returns "none" or "config" or "module" or "both" or "config" when not set

Properties

The following properties affect the CMakeDeps generator:

- **cmake_file_name**: The config file generated for the current package will follow the `<VALUE>-config.cmake` pattern, so to find the package you write `find_package(<VALUE>)`.
- **cmake_target_name**: Name of the target to be consumed.
- **cmake_target_aliases**: List of aliases that Conan will create for an already existing target.
- **cmake_find_mode**: Defaulted to `config`. Possible values are:
  - `config`: The CMakeDeps generator will create config scripts for the dependency.
  - `module`: Will create module config (FindXXX.cmake) scripts for the dependency.
  - `both`: Will generate both config and modules.
  - `none`: Won't generate any file. It can be used, for instance, to create a system wrapper package so the consumers find the config files in the CMake installation config path and not in the generated by Conan (because it has been skipped).
- **cmake_module_file_name**: Same as **cmake_file_name** but when generating modules with `cmake_find_mode=module/both`. If not specified it will default to **cmake_file_name**.
- **cmake_module_target_name**: Same as **cmake_target_name** but when generating modules with `cmake_find_mode=module/both`. If not specified it will default to **cmake_target_name**.
- **cmake_build_modules**: List of `.cmake` files (route relative to root package folder) that are automatically included when the consumer run the `find_package()`. This property cannot be set in the components, only in the root `self.cpp_info`.
- **cmake_set_interface_link_directories**: boolean value that should be only used by dependencies that don't declare *self.cpp_info.libs* but have `#pragma comment(lib, "foo")` (automatic link) declared at the public headers. Those dependencies should add this property to their *conanfile.py* files at root `cpp_info` level (components not supported for now).
- **nosoname**: boolean value that should be used only by dependencies that are defined as `SHARED` and represent a library built without the `soname` flag option.

Example:

```
def package_info(self):
    ...
    # MyFileName-config.cmake
    self.cpp_info.set_property("cmake_file_name", "MyFileName")
    # Names for targets are absolute, Conan won't add any namespace to the target␣
↪names automatically
    self.cpp_info.set_property("cmake_target_name", "Foo::Foo")
    # Automatically include the lib/mypkg.cmake file when calling find_package()
    # This property cannot be set in a component.
    self.cpp_info.set_property("cmake_build_modules", [os.path.join("lib", "mypkg.
↪cmake")])

    # Create a new target "MyFooAlias" that is an alias to the "Foo::Foo" target
    self.cpp_info.set_property("cmake_target_aliases", ["MyFooAlias"])

    self.cpp_info.components["mycomponent"].set_property("cmake_target_name",
↪"Foo::Var")

    # Create a new target "VarComponent" that is an alias to the "Foo::Var" component␣
↪target
    self.cpp_info.components["mycomponent"].set_property("cmake_target_aliases", [
↪"VarComponent"])

    # Skip this package when generating the files for the whole dependency tree in␣
↪the consumer
    # note: it will make useless the previous adjustements.
    # self.cpp_info.set_property("cmake_find_mode", "none")

    # Generate both MyFileNameConfig.cmake and FindMyFileName.cmake
    self.cpp_info.set_property("cmake_find_mode", "both")
```

Overwrite properties from the consumer side using CMakeDeps.set_property()

Using `CMakeDeps.set_property()` method you can overwrite the property values set by the Conan recipes from the consumer. This can be done for *cmake_file_name*, *cmake_target_name*, *cmake_find_mode*, *cmake_module_file_name* and *cmake_module_target_name* properties. Let's see an example of how this works:

Imagine we have a *compressor/1.0* package that depends on *zlib/1.2.11*. The *zlib* recipe defines some properties:

Listing 16: Zlib conanfile.py

```
class ZlibConan(ConanFile):
    name = "zlib"

    ...

    def package_info(self):
        self.cpp_info.set_property("cmake_find_mode", "both")
        self.cpp_info.set_property("cmake_file_name", "ZLIB")
        self.cpp_info.set_property("cmake_target_name", "ZLIB::ZLIB")
        ...
```

This recipe defines several properties. For example the `cmake_find_mode` property is set to `both`. That means that module and config files are generated for Zlib. Maybe we need to alter this behaviour and just generate config files. You could do that in the compressor recipe using the `CMakeDeps.set_property()` method:

Listing 17: compressor conanfile.py

```
class Compressor(ConanFile):
    name = "compressor"

    requires = "zlib/1.2.11"
    ...

    def generate(self):
        deps = CMakeDeps(self)
        deps.set_property("zlib", "cmake_find_mode", "config")
        deps.generate()
        ...
```

You can also use the `set_property()` method to invalidate the property values set by the upstream recipe and use the values that Conan assigns by default. To do so, set the value `None` to the property like this:

Listing 18: compressor conanfile.py

```
class Compressor(ConanFile):
    name = "compressor"

    requires = "zlib/1.2.11"
    ...

    def generate(self):
        deps = CMakeDeps(self)
        deps.set_property("zlib", "cmake_target_name", None)
        deps.generate()
        ...
```

After doing this the generated target name for the Zlib library will be `zlib::zlib` instead of `ZLIB::ZLIB`

Disable CMakeDeps For Installed CMake configuration files

Some projects may want to disable the `CMakeDeps` generator for downstream consumers. This can be done by settings `cmake_find_mode` to `none`. If the project wants to provide its own configuration targets, it should append them to the `buildirs` attribute of `cpp_info`.

This method is intended to work with downstream consumers using the `CMakeToolchain` generator, which will be populated with the `builddirs` attribute.

Example:

```
def package(self):
    ...
    cmake.install()

def package_info(self):
    self.cpp_info.set_property("cmake_find_mode", "none") # Do NOT generate anyfiles
    self.cpp_info.builddirs.append(os.path.join("lib", "cmake", "foo"))
```

Map from project configuration to imported target's configuration

As mentioned above, CMakeDeps provides support for multiple configuration environments (Debug, Release, etc.) This is achieved by populating properties on the imported targets according to the build_type setting when installing dependencies. When a consumer project is configured with a single-configuration CMake generator, however, it is necessary to define the CMAKE_BUILD_TYPE with a value that matches that of the installed dependencies.

If the consumer CMake project is configured with a different build type than the dependencies, it is necessary to tell CMake how to map the configurations from the current project to the imported targets by setting the CMAKE_MAP_IMPORTED_CONFIG_<CONFIG> CMake variable.

```
cd build-coverage/
conan install .. -s build_type=Debug
cmake .. -DCMAKE_BUILD_TYPE=Coverage -DCMAKE_TOOLCHAIN_FILE=conan_toolchain.cmake -
↪DCMAKE_MAP_IMPORTED_CONFIG_COVERAGE=Debug
```

CMakeToolchain

The CMakeToolchain is the toolchain generator for CMake. It produces the toolchain file that can be used in the command line invocation of CMake with the -DCMAKE_TOOLCHAIN_FILE=conan_toolchain.cmake. This generator translates the current package configuration, settings, and options, into CMake toolchain syntax.

It can be declared as:

```python
from conan import ConanFile

class Pkg(ConanFile):
    generators = "CMakeToolchain"
```

Or fully instantiated in the generate() method:

```python
from conan import ConanFile
from conan.tools.cmake import CMakeToolchain

class App(ConanFile):
    settings = "os", "arch", "compiler", "build_type"
    requires = "hello/0.1"
    generators = "CMakeDeps"
    options = {"shared": [True, False], "fPIC": [True, False]}
    default_options = {"shared": False, "fPIC": True}

    def generate(self):
        tc = CMakeToolchain(self)
        tc.variables["MYVAR"] = "MYVAR_VALUE"
        tc.preprocessor_definitions["MYDEFINE"] = "MYDEF_VALUE"
        tc.generate()
```

Note: The CMakeToolchain is intended to run with the CMakeDeps dependencies generator. Please do not use other CMake legacy generators (like cmake, or cmake_paths) with it.

Generated files

This will generate the following files after a `conan install` (or when building the package in the cache) with the information provided in the `generate()` method as well as information translated from the current `settings`:

- **conan_toolchain.cmake**: containing the translation of Conan settings to CMake variables. Some things that will be defined in this file:
  - Definition of the CMake generator platform and generator toolset
  - Definition of the `CMAKE_POSITION_INDEPENDENT_CODE`, based on `fPIC` option.
  - Definition of the C++ standard as necessary
  - Definition of the standard library used for C++
  - Deactivation of rpaths in OSX
- **conanvcvars.bat**: In some cases, the Visual Studio environment needs to be defined correctly for building, like when using the Ninja or NMake generators. If necessary, the `CMakeToolchain` will generate this script, so defining the correct Visual Studio prompt is easier.
- **CMakePresets.json**: The toolchain also generates a `CMakePresets.json` standard file, check the documentation here. It is currently using the version "3" of the JSON schema. Conan creates a `conan-default` configure preset with the information:
  - The `generator` to be used.
  - The path to the `conan_toolchain.cmake`.
  - Some cache variables corresponding to the specified settings cannot work if specified in the toolchain.
  - The `CMAKE_BUILD_TYPE` variable when using a single-configuration generators.
  - The `BUILD_TESTING` variable set to `OFF`, when configuration `tools.build:skip_test` is true.
  - If you run several `conan install` with different `-s build_type` values, it will generate the corresponding `buildPresets` and `configurePresets`.
  - By default, the presets names will be `conan-xxxx`, but the "conan-" prefix can be customized with `CMakeToolchain.presets_prefix = "conan"` attribute.
  - The preset names will be controlled by the `layout() self.folders.build_folder_vars` definition, that can contain a list of settings and options like `["settings.compiler", "settings.arch", "options.shared"]`.
- **CMakeUserPresets.json**: If you declare a `layout()` in the recipe and your `CMakeLists.txt` file is found at the `conanfile.source_folder` folder, a `CMakeUserPresets.json` file will be generated (if doesn't exist already) including automatically the `CMakePresets.json` (at the `conanfile.generators_folder`) to allow your IDE (Visual Studio, Visual Studio Code, CLion...) or `cmake` tool to locate the `CMakePresets.json`. The location of the generated `CMakeUserPresets.json` can be further tweaked by the `user_presets_path` attribute, as documented below. The version schema of the generated `CMakeUserPresets.json` is "4" and requires CMake >= 3.23. The file name of this file can be configured with the `CMakeToolchain.user_presets_path = "CMakeUserPresets.json"`` attribute, so if you want to generate a "ConanPresets.json" instead to be included from your own file, you can define `tc.user_presets_path = "ConanPresets.jon"` in the `generate()` method. See *extending your own CMake presets* for a full example.

  Note: Conan will skip the generation of the `CMakeUserPresets.json` if it already exists and was not generated by Conan.

Note: The version schema of the generated `CMakeUserPresets.json` is 4 (compatible with CMake>=3.23) and the schema for the `CMakePresets.json` is 3 (compatible with CMake>=3.21).

Customization

preprocessor_definitions

This attribute allows defining compiler preprocessor definitions, for multiple configurations (Debug, Release, etc).

```
def generate(self):
    tc = CMakeToolchain(self)
    tc.preprocessor_definitions["MYDEF"] = "MyValue"
    tc.preprocessor_definitions.debug["MYCONFIGDEF"] = "MyDebugValue"
    tc.preprocessor_definitions.release["MYCONFIGDEF"] = "MyReleaseValue"
    tc.generate()
```

This will be translated to:

- One `add_definitions()` definition for `MYDEF` in `conan_toolchain.cmake` file.
- One `add_definitions()` definition, using a cmake generator expression in `conan_toolchain.cmake` file, using the different values for different configurations.

cache_variables

This attribute allows defining CMake cache-variables. These variables, unlike the `variables`, are single-config. They will be stored in the `CMakePresets.json` file (at the *cacheVariables* in the *configurePreset*) and will be applied with `-D` arguments when calling `cmake.configure` using the *CMake() build helper*.

```
def generate(self):
    tc = CMakeToolchain(self)
    tc.cache_variables["foo"] = True
    tc.cache_variables["foo2"] = False
    tc.cache_variables["var"] = "23"
```

The booleans assigned to a cache_variable will be translated to `ON` and `OFF` symbols in CMake.

variables

This attribute allows defining CMake variables, for multiple configurations (Debug, Release, etc). These variables should be used to define things related to the toolchain and for the majority of cases *cache_variables* is what you probably want to use. Also, take into account that as these variables are defined inside the *conan_toolchain.cmake* file, and the toolchain is loaded several times by CMake, the definition of these variables will be done at those points as well.

```
def generate(self):
    tc = CMakeToolchain(self)
    tc.variables["MYVAR"] = "MyValue"
    tc.variables.debug["MYCONFIGVAR"] = "MyDebugValue"
    tc.variables.release["MYCONFIGVAR"] = "MyReleaseValue"
    tc.generate()
```

This will be translated to:

- One `set()` definition for `MYVAR` in `conan_toolchain.cmake` file.
- One `set()` definition, using a cmake generator expression in `conan_toolchain.cmake` file, using the different values for different configurations.

The booleans assigned to a variable will be translated to `ON` and `OFF` symbols in CMake:

```
def generate(self):
    tc = CMakeToolchain(self)
    tc.variables["FOO"] = True
    tc.variables["VAR"] = False
    tc.generate()
```

Will generate the sentences: `set(FOO ON ...)` and `set(VAR OFF ...)`.

user_presets_path

This attribute allows specifying the location of the generated `CMakeUserPresets.json` file. Accepted values:

- An absolute path
- A path relative to `self.source_folder`
- The boolean value `False`, to suppress the generation of the file altogether.

For example, we can prevent the generator from creating `CMakeUserPresets.json` in the following way:

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

presets_prefix

By default it is `"conan"`, and it will generate CMake presets named "conan-xxxx". This is done to avoid potential name clashes with users own presets.

Using a custom toolchain file

There are two ways of providing custom CMake toolchain files:

- The `conan_toolchain.cmake` file can be completely skipped and replaced by a user one, defining the `tools.cmake.cmaketoolchain:toolchain_file=<filepath>` configuration value.
- A custom user toolchain file can be added (included from) to the `conan_toolchain.cmake` one, by using the `user_toolchain` block described below, and defining the `tools.cmake.cmaketoolchain:user_toolchain=["<filepath>"]` configuration value.

  The configuration `tools.cmake.cmaketoolchain:user_toolchain=["<filepath>"]` can be defined in the `global.conf`. but also creating a Conan package for your toolchain and using `self.conf_info` to declare the toolchain file:

```
import os
from conan import ConanFile
class MyToolchainPackage(ConanFile):
    ...
    def package_info(self):
        f = os.path.join(self.package_folder, "mytoolchain.cmake")
        self.conf_info.define("tools.cmake.cmaketoolchain:user_toolchain",
↪ [f])
```

If you declare the previous package as a `tool_require`, the toolchain will be automatically applied.

- If you have more than one `tool_requires` defined, you can easily append all the user toolchain values together using the `append` method in each of them, for instance:

```
import os
from conan import ConanFile
class MyToolRequire(ConanFile):
    ...
    def package_info(self):
        f = os.path.join(self.package_folder, "mytoolchain.cmake")
        # Appending the value to any existing one
        self.conf_info.append("tools.cmake.cmaketoolchain:user_toolchain",
↪ f)
```

So, they'll be automatically applied by your `CMakeToolchain` generator without writing any extra code:

```
from conan import ConanFile
from conan.tools.cmake import CMake
class Pkg(ConanFile):
    settings = "os", "compiler", "arch", "build_type"
    exports_sources = "CMakeLists.txt"
    tool_requires = "toolchain1/0.1", "toolchain2/0.1"
    generators = "CMakeToolchain"

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

Extending and advanced customization

`CMakeToolchain` implements a powerful capability for extending and customizing the resulting toolchain file.

The contents are organized by `blocks` that can be customized. The following predefined blocks are available, and added in this order:

- **user_toolchain**: Allows to include user toolchains from the `conan_toolchain.cmake` file. If the configuration `tools.cmake.cmaketoolchain:user_toolchain=["xxxx", "yyyy"]` is defined, its values will be `include(xxx)\ninclude(yyyy)` as the first lines in `conan_toolchain.cmake`.
- **generic_system**: Defines `CMAKE_SYSTEM_NAME`, `CMAKE_SYSTEM_VERSION`, `CMAKE_SYSTEM_PROCESSOR`, `CMAKE_GENERATOR_PLATFORM`, `CMAKE_GENERATOR_TOOLSET`, `CMAKE_C_COMPILER`, `CMAKE_CXX_COMPILER`
- **android_system**: Defines `ANDROID_PLATFORM`, `ANDROID_STL`, `ANDROID_ABI` and includes `ANDROID_NDK_PATH/build/cmake/android.toolchain.cmake` where `ANDROID_NDK_PATH` comes defined in `tools.android:ndk_path` configuration value.
- **apple_system**: Defines `CMAKE_OSX_ARCHITECTURES`, `CMAKE_OSX_SYSROOT` for Apple systems.

- **fpic**: Defines the `CMAKE_POSITION_INDEPENDENT_CODE` when there is a `options.fPIC`
- **arch_flags**: Defines C/C++ flags like `-m32, -m64` when necessary.
- **linker_scripts**: Defines the flags for any provided linker scripts.
- **libcxx**: Defines `-stdlib=libc++` flag when necessary as well as `_GLIBCXX_USE_CXX11_ABI`.
- **vs_runtime**: Defines the `CMAKE_MSVC_RUNTIME_LIBRARY` variable, as a generator expression for multiple configurations.
- **cppstd**: defines `CMAKE_CXX_STANDARD`, `CMAKE_CXX_EXTENSIONS`
- **parallel**: defines `/MP` parallel build flag for Visual.
- **cmake_flags_init**: defines `CMAKE_XXX_FLAGS` variables based on previously defined Conan variables. The blocks above only define `CONAN_XXX` variables, and this block will define CMake ones like `set(CMAKE_CXX_FLAGS_INIT "${CONAN_CXX_FLAGS}" CACHE STRING "" FORCE)``.
- **try_compile**: Stop processing the toolchain, skipping the blocks below this one, if `IN_TRY_COMPILE` CMake property is defined.
- **find_paths**: Defines `CMAKE_FIND_PACKAGE_PREFER_CONFIG`, `CMAKE_MODULE_PATH`, `CMAKE_PREFIX_PATH` so the generated files from `CMakeDeps` are found.
- **rpath**: Defines `CMAKE_SKIP_RPATH`. By default it is disabled, and it is needed to define `self.blocks["rpath"].skip_rpath=True` if you want to activate `CMAKE_SKIP_RPATH`
- **shared**: defines `BUILD_SHARED_LIBS`.
- **output_dirs**: Define the `CMAKE_INSTALL_XXX` variables.
  - **CMAKE_INSTALL_PREFIX**: Is set with the `package_folder`, so if a "cmake install" operation is run, the artifacts go to that location.
  - **CMAKE_INSTALL_BINDIR**, **CMAKE_INSTALL_SBINDIR** and **CMAKE_INSTALL_LIBEXECDIR**: Set by default to `bin`.
  - **CMAKE_INSTALL_LIBDIR**: Set by default to `lib`.
  - **CMAKE_INSTALL_INCLUDEDIR** and **CMAKE_INSTALL_OLDINCLUDEDIR**: Set by default to `include`.
  - **CMAKE_INSTALL_DATAROOTDIR**: Set by default to `res`.

If you want to change the default values, adjust the `cpp.package` object at the `layout()` method:

```
def layout(self):
    ...
    # For CMAKE_INSTALL_BINDIR, CMAKE_INSTALL_SBINDIR and CMAKE_
↪INSTALL_LIBEXECDIR, takes the first value:
    self.cpp.package.bindirs = ["mybin"]
    # For CMAKE_INSTALL_LIBDIR, takes the first value:
    self.cpp.package.libdirs = ["mylib"]
    # For CMAKE_INSTALL_INCLUDEDIR, CMAKE_INSTALL_OLDINCLUDEDIR,␣
↪takes the first value:
    self.cpp.package.includedirs = ["myinclude"]
    # For CMAKE_INSTALL_DATAROOTDIR, takes the first value:
    self.cpp.package.resdirs = ["myres"]
```

Note: It is **not valid** to change the self.cpp_info at the `package_info()` method.

Customizing the content blocks

Every block can be customized in different ways:

```
# remove an existing block
def generate(self):
    tc = CMakeToolchain(self)
    tc.blocks.remove("generic_system")

# modify the template of an existing block
def generate(self):
    tc = CMakeToolchain(self)
    tmp = tc.blocks["generic_system"].template
    new_tmp = tmp.replace(...)  # replace, fully replace, append...
    tc.blocks["generic_system"].template = new_tmp

# modify one or more variables of the context
def generate(self):
    tc = CMakeToolchain(conanfile)
    # block.values is the context dictionary
    toolset = tc.blocks["generic_system"].values["toolset"]
    tc.blocks["generic_system"].values["toolset"] = "other_toolset"

# modify the whole context values
def generate(self):
    tc = CMakeToolchain(conanfile)
    tc.blocks["generic_system"].values = {"toolset": "other_toolset"}

# modify the context method of an existing block
import types

def generate(self):
    tc = CMakeToolchain(self)
    generic_block = toolchain.blocks["generic_system"]

    def context(self):
        assert self  # Your own custom logic here
        return {"toolset": "other_toolset"}
    generic_block.context = types.MethodType(context, generic_block)

# completely replace existing block
from conan.tools.cmake import CMakeToolchain

def generate(self):
    tc = CMakeToolchain(self)
    # this could go to a python_requires
    class MyGenericBlock:
        template = "HelloWorld"

        def context(self):
            return {}

    tc.blocks["generic_system"] = MyGenericBlock

# add a completely new block
from conan.tools.cmake import CMakeToolchain
def generate(self):
```

(continues on next page)

(continued from previous page)

```
    tc = CMakeToolchain(self)
    # this could go to a python_requires
    class MyBlock:
        template = "Hello {{myvar}}!!!"

        def context(self):
            return {"myvar": "World"}

    tc.blocks["mynewblock"] = MyBlock
```

For more information about these blocks, please have a look at the source code.

Cross building

The `generic_system` block contains some basic cross-building capabilities. In the general case, the user would want to provide their own user toolchain defining all the specifics, which can be done with the configuration `tools.cmake.cmaketoolchain:user_toolchain`. If this conf value is defined, the `generic_system` block will include the provided file or files, but no further define any CMake variable for cross-building.

If `user_toolchain` is not defined and Conan detects it is cross-building, because the build and host profiles contain different OS or architecture, it will try to define the following variables:

- `CMAKE_SYSTEM_NAME`: `tools.cmake.cmaketoolchain:system_name` configuration if defined, otherwise, it will try to autodetect it. This block will consider cross-building if Android systems (that is managed by other blocks), and not 64bits to 32bits builds in x86_64, sparc and ppc systems.
- `CMAKE_SYSTEM_VERSION`: `tools.cmake.cmaketoolchain:system_version` conf if defined, otherwise `os.version` subsetting (host) when defined
- `CMAKE_SYSTEM_PROCESSOR`: `tools.cmake.cmaketoolchain:system_processor` conf if defined, otherwise `arch` setting (host) if defined

Reference

class CMakeToolchain(*conanfile*, *generator=None*)

generate()
This method will save the generated files to the conanfile.generators_folder

conf

CMakeToolchain is affected by these `[conf]` variables:

- **tools.cmake.cmaketoolchain:toolchain_file** user toolchain file to replace the `conan_toolchain.cmake` one.
- **tools.cmake.cmaketoolchain:user_toolchain** list of user toolchains to be included from the `conan_toolchain.cmake` file.
- **tools.android:ndk_path** value for `ANDROID_NDK_PATH`.
- **tools.cmake.cmaketoolchain:system_name** is not necessary in most cases and is only used to force-define `CMAKE_SYSTEM_NAME`.

- **tools.cmake.cmaketoolchain:system_version** is not necessary in most cases and is only used to force-define `CMAKE_SYSTEM_VERSION`.
- **tools.cmake.cmaketoolchain:system_processor** is not necessary in most cases and is only used to force-define `CMAKE_SYSTEM_PROCESSOR`.
- **tools.cmake.cmaketoolchain:toolset_arch**: Will add the `,host=xxx` specifier in the `CMAKE_GENERATOR_TOOLSET` variable of `conan_toolchain.cmake` file.
- **tools.cmake.cmake_layout:build_folder_vars**: Settings and Options that will produce a different build folder and different CMake presets names.
- **tools.build:cxxflags** list of extra C++ flags that will be appended to `CMAKE_CXX_FLAGS_INIT`.
- **tools.build:cflags** list of extra of pure C flags that will be appended to `CMAKE_C_FLAGS_INIT`.
- **tools.build:sharedlinkflags** list of extra linker flags that will be appended to `CMAKE_SHARED_LINKER_FLAGS_INIT`.
- **tools.build:exelinkflags** list of extra linker flags that will be appended to `CMAKE_EXE_LINKER_FLAGS_INIT`.
- **tools.build:defines** list of preprocessor definitions that will be used by `add_definitions()`.
- **tools.build:tools.apple:enable_bitcode** boolean value to enable/disable Bitcode Apple Clang flags, e.g., `CMAKE_XCODE_ATTRIBUTE_ENABLE_BITCODE`.
- **tools.build:tools.apple:enable_arc** boolean value to enable/disable ARC Apple Clang flags, e.g., `CMAKE_XCODE_ATTRIBUTE_CLANG_ENABLE_OBJC_ARC`.
- **tools.build:tools.apple:enable_visibility** boolean value to enable/disable Visibility Apple Clang flags, e.g., `CMAKE_XCODE_ATTRIBUTE_GCC_SYMBOLS_PRIVATE_EXTERN`.
- **tools.build:sysroot** defines the value of `CMAKE_SYSROOT`.
- **tools.build:compiler_executables** dict-like Python object which specifies the compiler as key and the compiler executable path as value. Those keys will be mapped as follows:
  - `c`: will set `CMAKE_C_COMPILER` in *conan_toolchain.cmake*.
  - `cpp`: will set `CMAKE_CXX_COMPILER` in *conan_toolchain.cmake*.
  - `RC`: will set `CMAKE_RC_COMPILER` in *conan_toolchain.cmake*.
  - `objc`: will set `CMAKE_OBJC_COMPILER` in *conan_toolchain.cmake*.
  - `objcpp`: will set `CMAKE_OBJCXX_COMPILER` in *conan_toolchain.cmake*.
  - `cuda`: will set `CMAKE_CUDA_COMPILER` in *conan_toolchain.cmake*.
  - `fortran`: will set `CMAKE_Fortran_COMPILER` in *conan_toolchain.cmake*.
  - `asm`: will set `CMAKE_ASM_COMPILER` in *conan_toolchain.cmake*.
  - `hip`: will set `CMAKE_HIP_COMPILER` in *conan_toolchain.cmake*.
  - `ispc`: will set `CMAKE_ISPC_COMPILER` in *conan_toolchain.cmake*.

CMake

The `CMake` build helper is a wrapper around the command line invocation of cmake. It will abstract the calls like `cmake --build . --config Release` into Python method calls. It will also add the argument `-DCMAKE_TOOLCHAIN_FILE=conan_toolchain.cmake` (from the generator `CMakeToolchain`) to the `configure()` call, as well as other possible arguments like `-DCMAKE_BUILD_TYPE=<config>`. The arguments that will be used are obtained from a generated `CMakePresets.json` file.

The helper is intended to be used in the `build()` method, to call CMake commands automatically when a package is being built directly by Conan (create, install)

```
from conan import ConanFile
from conan.tools.cmake import CMake, CMakeToolchain, CMakeDeps

class App(ConanFile):
    settings = "os", "arch", "compiler", "build_type"
    requires = "hello/0.1"
    options = {"shared": [True, False], "fPIC": [True, False]}
    default_options = {"shared": False, "fPIC": True}

    def generate(self):
        tc = CMakeToolchain(self)
        tc.generate()
        deps = CMakeDeps(self)
        deps.generate()

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

Reference

class CMake(*conanfile*)

CMake helper to use together with the CMakeToolchain feature

Parameters **conanfile** – The current recipe object. Always use `self`.

configure(*variables=None*, *build_script_folder=None*, *cli_args=None*)

Reads the `CMakePresets.json` file generated by the :param cli_args: Extra CLI arguments to pass to cmake invocation *CMakeToolchain* to get:

- The generator, to append `-G="xxx"`.
- The path to the toolchain and append `-DCMAKE_TOOLCHAIN_FILE=/path/conan_toolchain.cmake`
- The declared `cache variables` and append `-Dxxx`.

and call `cmake`.

Parameters

- **variables** – Should be a dictionary of CMake variables and values, that will be mapped to command line `-DVAR=VALUE` arguments. Recall that in the general case information to CMake should be passed in `CMakeToolchain` to be provided in the `conan_toolchain.cmake` file. This `variables` argument is intended for exceptional cases that wouldn't work in the toolchain approach.
- **build_script_folder** – Path to the CMakeLists.txt in case it is not in the declared `self.folders.source` at the `layout()` method.
- **cli_args** – List of extra arguments provided when calling to CMake.

build(*build_type=None*, *target=None*, *cli_args=None*, *build_tool_args=None*)

Parameters

- **build_type** – Use it only to override the value defined in the `settings.build_type` for a multi-configuration generator (e.g. Visual Studio, XCode). This value will be ignored for single-configuration generators, they will use the one defined in the toolchain file during the install step.
- **target** – Name of the build target to run
- **cli_args** – A list of arguments `[arg1, arg2, ...]` that will be passed to the `cmake --build ... arg1 arg2` command directly.
- **build_tool_args** – A list of arguments `[barg1, barg2, ...]` for the underlying build system that will be passed to the command line after the `--` indicator: `cmake --build ... -- barg1 barg2`

install (*build_type=None*, *component=None*)
Equivalent to run `cmake --build . --target=install`

Parameters

- **component** – The specific component to install, if any
- **build_type** – Use it only to override the value defined in the settings.build_type. It can fail if the build is single configuration (e.g. Unix Makefiles), as in that case the build type must be specified at configure time, not build type.

test (*build_type=None*, *target=None*, *cli_args=None*, *build_tool_args=None*, *env=''*)
Equivalent to running cmake –build . –target=RUN_TESTS.

Parameters

- **build_type** – Use it only to override the value defined in the `settings.build_type`. It can fail if the build is single configuration (e.g. Unix Makefiles), as in that case the build type must be specified at configure time, not build time.
- **target** – Name of the build target to run, by default `RUN_TESTS` or `test`
- **cli_args** – Same as above `build()`
- **build_tool_args** – Same as above `build()`

conf

CMake() helper is affected by these `[conf]` variables:

- `tools.microsoft.msbuild:verbosity` will accept one of `"Quiet"`, `"Minimal"`, `"Normal"`, `"Detailed"`, `"Diagnostic"` to be passed to the `CMake.build()` command, when a Visual Studio generator (MSBuild build system) is being used for CMake. It is passed as an argument to the underlying build system via the call `cmake --build . --config Release -- /verbosity:Diagnostic`
- `tools.build:jobs` argument for the `--jobs` parameter when running Ninja generator.
- `tools.microsoft.msbuild:max_cpu_count` argument for the `/m` (`/maxCpuCount`) when running `MSBuild`

cmake_layout

The `cmake_layout()` sets the *folders* and *cpp* attributes to follow the structure of a typical CMake project.

```
from conan.tools.cmake import cmake_layout

def layout(self):
    cmake_layout(self)
```

Note: To try it you can use the `conan new -d name=hello -d version=1.0 cmake_lib` template.

The assigned values depend on the CMake generator that will be used. It can be defined with the `tools.cmake.cmaketoolchain:generator` [conf] entry or passing it in the recipe to the `cmake_layout(self, cmake_generator)` function. The assigned values are different if it is a multi-config generator (like Visual Studio or Xcode), or a single-config generator (like Unix Makefiles).

These are the values assigned by the `cmake_layout`:

- `conanfile.folders.source`: *src_folder* argument or `.` if not specified.
- **`conanfile.folders.build`:**
  - `build`: if the cmake generator is multi-configuration.
  - `build/Debug` or `build/Release`: if the cmake generator is single-configuration, depending on the build_type.
  - The `"build"` string, can be defined to other value by the `build_folder` argument.
- `conanfile.folders.generators`: `build/generators`
- `conanfile.cpp.source.includedirs`: `["include"]`
- **`conanfile.cpp.build.libdirs` and `conanfile.cpp.build.bindirs`:**
  - `["Release"]` or `["Debug"]` for a multi-configuration cmake generator.
  - `.` for a single-configuration cmake generator.

Reference

`cmake_layout`(*conanfile*, *generator=None*, *src_folder='.'*, *build_folder='build'*)

Parameters

- **`conanfile`** – The current recipe object. Always use `self`.
- **`generator`** – Allow defining the CMake generator. In most cases it doesn't need to be passed, as it will get the value from the configuration `tools.cmake.cmaketoolchain:generator`, or it will automatically deduce the generator from the `settings`
- **`src_folder`** – Value for `conanfile.folders.source`, change it if your source code (and CMakeLists.txt) is in a subfolder.
- **`build_folder`** – Specify the name of the "base" build folder. The default is "build", but if that folder name is used by the project, a different one can be defined

Multi-setting/option cmake_layout

The `folders.build` and `conanfile.folders.generators` can be customized to take into account the `settings` and `options` and not only the `build_type`. Use the `tools.cmake.cmake_layout:build_folder_vars` conf to declare a list of settings or options:

```
conan install . -c tools.cmake.cmake_layout:build_folder_vars="['settings.compiler',
↪'options.shared']"
```

For the previous example, the values assigned by the `cmake_layout` (installing the Release/static default configuration) would be:

- **conanfile.folders.build:**
  - `build/apple-clang-shared_false`: if the cmake generator is multi-configuration.
  - `build/apple-clang-shared_false/Debug`: if the cmake generator is single-configuration.
- `conanfile.folders.generators`: `build/generators`

If we repeat the previous install with a different configuration:

```
conan install . -o shared=True -c tools.cmake.cmake_layout:build_folder_vars="[
↪'settings.compiler', 'options.shared']"
```

The values assigned by the `cmake_layout` (installing the Release/shared configuration) would be:

- **conanfile.folders.build:**
  - `build/apple-clang-shared_true`: if the cmake generator is multi-configuration.
  - `build/apple-clang-shared_true/Debug`: if the cmake generator is single-configuration.
- `conanfile.folders.generators`: `build-apple-clang-shared_true/generators`

So we can keep separated folders for any number of different configurations that we want to install.

The `CMakePresets.json` file generated at the *CMakeToolchain* generator, will also take this `tools.cmake.cmake_layout:build_folder_vars` config into account to generate different names for the presets, being very handy to install N configurations and building our project for any of them by selecting the chosen preset.

7.4.2 conan.tools.gnu

AutotoolsDeps

The `AutotoolsDeps` is the dependencies generator for Autotools. It will generate shell scripts containing environment variable definitions that the autotools build system can understand.

It can be used by name in conanfiles:

Listing 19: conanfile.py

```
class Pkg(ConanFile):
    generators = "AutotoolsDeps"
```

Listing 20: conanfile.txt

```
[generators]
AutotoolsDeps
```

And it can also be fully instantiated in the conanfile `generate()` method:

```
from conan import ConanFile
from conan.tools.gnu import AutotoolsDeps

class App(ConanFile):
    settings = "os", "arch", "compiler", "build_type"

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

Generated files

It will generate the file `conanautotoolsdeps.sh` or `conanautotoolsdeps.bat`:

```
$ conan install conanfile.py # default is Release
$ source conanautotoolsdeps.sh
# or in Windows
$ conanautotoolsdeps.bat
```

These launchers will define aggregated variables `CPPFLAGS`, `LIBS`, `LDFLAGS`, `CXXFLAGS`, `CFLAGS` that accumulate all dependencies information, including transitive dependencies, with flags like `-I<path>`, `-L<path>`, etc.

At this moment, only the `requires` information is generated, the `tool_requires` one is not managed by this generator yet.

Customization

To modify the computed values, you can access the `.environment` property that returns an *Environment* class.

```
from conan import ConanFile
from conan.tools.gnu import AutotoolsDeps

class App(ConanFile):
    settings = "os", "arch", "compiler", "build_type"

    def generate(self):
        tc = AutotoolsDeps(self)
        tc.environment.remove("CPPFLAGS", "undesired_value")
        tc.environment.append("CPPFLAGS", "var")
        tc.environment.define("OTHER", "cat")
        tc.environment.unset("LDFLAGS")
        tc.generate()
```

Reference

class AutotoolsDeps(*conanfile*)

environment

Returns An `Environment` object containing the computed variables. If you need to modify some of the computed values you can access to the `environment` object.

AutotoolsToolchain

The `AutotoolsToolchain` is the toolchain generator for Autotools. It will generate shell scripts containing environment variable definitions that the autotools build system can understand.

This generator can be used by name in conanfiles:

Listing 21: conanfile.py

```
class Pkg(ConanFile):
    generators = "AutotoolsToolchain"
```

Listing 22: conanfile.txt

```
[generators]
AutotoolsToolchain
```

And it can also be fully instantiated in the conanfile `generate()` method:

```
from conan import ConanFile
from conan.tools.gnu import AutotoolsToolchain

class App(ConanFile):
    settings = "os", "arch", "compiler", "build_type"

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

Generated files

It will generate the file `conanautotoolstoolchain.sh` or `conanautotoolstoolchain.bat` files:

```
$ conan install conanfile.py # default is Release
$ source conanautotoolstoolchain.sh
# or in Windows
$ conanautotoolstoolchain.bat
```

This launchers will append information to the `CPPFLAGS`, `LDFLAGS`, `CXXFLAGS`, `CFLAGS` environment variables that translate the settings and options to the corresponding build flags like `-stdlib=libstdc++`, `-std=gnu14`, architecture flags, etc. It will also append the folder where the Conan generators are located to the `PKG_CONFIG_PATH` environment variable.

This generator will also generate a file called `conanbuild.conf` containing two keys:

- **configure_args**: Arguments to call the `configure` script.

- **make_args**: Arguments to call the `make` script.
- **autoreconf_args**: Arguments to call the `autoreconf` script.

The *Autotools build helper* will use that `conanbuild.conf` file to seamlessly call the configure and make script using these precalculated arguments.

Customization

You can change some attributes before calling the `generate()` method if you want to change some of the precalculated values:

```
from conan import ConanFile
from conan.tools.gnu import AutotoolsToolchain

class App(ConanFile):
    settings = "os", "arch", "compiler", "build_type"

    def generate(self):
        tc = AutotoolsToolchain(self)
        tc.configure_args.append("--my_argument")
        tc.generate()
```

- **configure_args: Additional arguments to be passed to the configure script.**
  - **By default the following arguments are passed:**
    - `--prefix`: Takes / as default value.
    - `--bindir=${prefix}/bin`
    - `--sbindir=${prefix}/bin`
    - `--libdir=${prefix}/lib`
    - `--includedir=${prefix}/include`
    - `--oldincludedir=${prefix}/include`
    - `--datarootdir=${prefix}/res`
  - **Also if the shared option exists it will add by default:**
    - `--enable-shared`, `--disable-static` if `shared==True`
    - `--disable-shared`, `--enable-static` if `shared==False`
- **make_args** (Defaulted to `[]`): Additional arguments to be passed to he make script.
- **autoreconf_args** (Defaulted to `["--force", "--install"]`): Additional arguments to be passed to he make script.
- **extra_defines** (Defaulted to `[]`): Additional defines.
- **extra_cxxflags** (Defaulted to `[]`): Additional cxxflags.
- **extra_cflags** (Defaulted to `[]`): Additional cflags.
- **extra_ldflags** (Defaulted to `[]`): Additional ldflags.
- **ndebug**: "NDEBUG" if the `settings.build_type` != *Debug*.
- **gcc_cxx11_abi**: "_GLIBCXX_USE_CXX11_ABI" if `gcc/libstdc++`.
- **libcxx**: Flag calculated from `settings.compiler.libcxx`.

- **fpic**: True/False from `options.fpic` if defined.
- **cppstd**: Flag from `settings.compiler.cppstd`
- **arch_flag**: Flag from `settings.arch`
- **build_type_flags**: Flags from `settings.build_type`
- **sysroot_flag**: To pass the `--sysroot` flag to the compiler.
- **apple_arch_flag**: Only when cross-building with Apple systems. Flags from `settings.arch`.
- **apple_isysroot_flag**: Only when cross-building with Apple systems. Path to the root sdk.
- **msvc_runtime_flag**: Flag from `settings.compiler.runtime_type` when compiler is `msvc` or `settings.compiler.runtime` when using the deprecated `Visual Studio`.

The following attributes are ready-only and will contain the calculated values for the current configuration and customized attributes. Some recipes might need to read them to generate custom build files (not strictly Autotools) with the configuration:

- **defines**
- **cxxflags**
- **cflags**
- **ldflags**

```
from conan import ConanFile
from conan.tools.gnu import AutotoolsToolchain

class App(ConanFile):
    settings = "os", "arch", "compiler", "build_type"
    def generate(self):
        tc = AutotoolsToolchain(self)
        # Customize the flags
        tc.extra_cxxflags = ["MyFlag"]
        # Read the computed flags and use them (write custom files etc)
        tc.defines
        tc.cxxflags
        tc.cflags
        tc.ldflags
```

If you want to change the default values for `configure_args`, adjust the `cpp.package` object at the `layout()` method:

```
def layout(self):
    ...
    # For bindir and sbindir takes the first value:
    self.cpp.package.bindirs = ["mybin"]
    # For libdir takes the first value:
    self.cpp.package.libdirs = ["mylib"]
    # For includedir and oldincludedir takes the first value:
    self.cpp.package.includedirs = ["myinclude"]
    # For datarootdir takes the first value:
    self.cpp.package.resdirs = ["myres"]
```

Note: It is **not valid** to change the self.cpp_info at the `package_info()` method.

Customizing the environment

If your `Makefile` or `configure` scripts need some other environment variable rather than `CPPFLAGS`, `LDFLAGS`, `CXXFLAGS` or `CFLAGS`, you can customize it before calling the `generate()` method. Call the `environment()` method to calculate the mentioned variables and then add the variables that you need. The `environment()` method returns an *Environment* object:

```
from conan import ConanFile
from conan.tools.gnu import AutotoolsToolchain

class App(ConanFile):
    settings = "os", "arch", "compiler", "build_type"

    def generate(self):
        at = AutotoolsToolchain(self)
        env = at.environment()
        env.define("FOO", "BAR")
        at.generate(env)
```

The `AutotoolsToolchain` also sets `CXXFLAGS`, `CFLAGS`, `LDFLAGS` and `CPPFLAGS` reading variables from the `[conf]` section in the profiles. *See the conf reference below*.

Managing the configure_args, make_args and autoreconf_args attributes

`AutotoolsToolchain` provides some help methods so users can add/update/remove values defined in `configure_args`, `make_args` and `autoreconf_args` (all of them lists of strings). Those methods are:

- `update_configure_args(updated_flags)`: will change `AutotoolsToolchain.configure_args`.
- `update_make_args(updated_flags)`: will change `AutotoolsToolchain.make_args`.
- `update_autoreconf_args(updated_flags)`: will change `AutotoolsToolchain.autoreconf_args`.

Where `updated_flags` is a dict-like Python object defining all the flags to change. It follows the next rules:

- Key-value are the flags names and their values, e.g., `{"--enable-tools": no}` will be translated as `--enable-tools=no`.
- If that key has no value, then it will be an empty string, e.g., `{"--disable-verbose": ""}` will be translated as `--disable-verbose`.
- If the key value is `None`, it means that you want to remove that flag from the `xxxxxx_args` (notice that it could be `configure_args`, `make_args` or `autoreconf_args`), e.g., `{"--force": None}` will remove that flag from the final result.

In a nutshell, you will:

- **Add arguments**: if the given flag in `updated_flags` does not already exist in `xxxxxx_args`.
- **Update arguments**: if the given flag in `updated_flags` already exists in attribute `xxxxxx_args`.
- **Remove arguments**: if the given flag in `updated_flags` already exists in `xxxxxx_args` and it's passed with `None` as value.

For instance:

```python
from conan import ConanFile
from conan.tools.gnu import AutotoolsToolchain

class App(ConanFile):
    settings = "os", "arch", "compiler", "build_type"

    def generate(self):
        at = AutotoolsToolchain(self)
        at.update_configure_args({
            "--new-super-flag": "",  # add new flag '--new-super-flag'
            "--host": "my-gnu-triplet",  # update flag '--host=my-gnu-triplet'
            "--force": None  # remove existing '--force' flag
        })
        at.generate()
```

Reference

class AutotoolsToolchain(*conanfile*, *namespace=None*, *prefix='/'*)

Parameters

- **conanfile** – The current recipe object. Always use `self`.
- **namespace** – This argument avoids collisions when you have multiple toolchain calls in the same recipe. By setting this argument, the *conanbuild.conf* file used to pass information to the build helper will be named as *<namespace>_conanbuild.conf*. The default value is `None` meaning that the name of the generated file is *conanbuild.conf*. This namespace must be also set with the same value in the constructor of the Autotools build helper so that it reads the information from the proper file.
- **prefix** – Folder to use for `--prefix` argument ("/" by default).

update_configure_args(*updated_flags*)

Helper to update/prune flags from `self.configure_args`.

Parameters **updated_flags** – `dict` with arguments as keys and their argument values. Notice that if argument value is `None`, this one will be pruned.

update_make_args(*updated_flags*)

Helper to update/prune arguments from `self.make_args`.

Parameters **updated_flags** – `dict` with arguments as keys and their argument values. Notice that if argument value is `None`, this one will be pruned.

update_autoreconf_args(*updated_flags*)

Helper to update/prune arguments from `self.autoreconf_args`.

Parameters **updated_flags** – `dict` with arguments as keys and their argument values. Notice that if argument value is `None`, this one will be pruned.

conf

- `tools.build:cxxflags` list of extra C++ flags that will be used by `CXXFLAGS`.
- `tools.build:cflags` list of extra of pure C flags that will be used by `CFLAGS`.
- `tools.build:sharedlinkflags` list of extra linker flags that will be used by `LDFLAGS`.
- `tools.build:exelinkflags` list of extra linker flags that will be used by `LDFLAGS`.

- `tools.build:linker_scripts` list of linker scripts, each of which will be prefixed with `-T` and added to `LDFLAGS`. Only use this flag with linkers that supports specifying linker scripts with the `-T` flag, such as `ld`, `gold`, and `lld`.
- `tools.build:defines` list of preprocessor definitions that will be used by `CPPFLAGS`.
- `tools.build:sysroot` defines the `--sysroot` flag to the compiler.
- `tools.build:compiler_executables` dict-like Python object which specifies the compiler as key and the compiler executable path as value. Those keys will be mapped as follows:
  - `c`: will set `CC` in *conanautotoolstoolchain.sh|bat* script.
  - `cpp`: will set `CXX` in *conanautotoolstoolchain.sh|bat* script.
  - `cuda`: will set `NVCC` in *conanautotoolstoolchain.sh|bat* script.
  - `fortran`: will set `FC` in *conanautotoolstoolchain.sh|bat* script.

Autotools

The `Autotools` build helper is a wrapper around the command line invocation of autotools. It will abstract the calls like `./configure` or `make` into Python method calls.

Usage:

```
from conan import ConanFile
from conan.tools.gnu import Autotools

class App(ConanFile):
    settings = "os", "arch", "compiler", "build_type"

    def build(self):
        autotools = Autotools(self)
        autotools.configure()
        autotools.make()
```

It will read the `conanbuild.conf` file generated by the *AutotoolsToolchain* to know read the arguments for calling the configure and make scripts:

- **configure_args**: Arguments to call the `configure` script.
- **make_args**: Arguments to call the `make` script.

Reference

class Autotools(*conanfile*, *namespace=None*)

Parameters

- **conanfile** – The current recipe object. Always use `self`.
- **namespace** – this argument avoids collisions when you have multiple toolchain calls in the same recipe. By setting this argument, the *conanbuild.conf* file used to pass information to the toolchain will be named as: *<namespace>_conanbuild.conf*. The default value is `None` meaning that the name of the generated file is *conanbuild.conf*. This namespace must be also set with the same value in the constructor of the AutotoolsToolchain so that it reads the information from the proper file.

configure(*build_script_folder=None*, *args=None*)
Call the configure script.

Parameters

- **args** – List of arguments to use for the `configure` call.
- **build_script_folder** – Subfolder where the *configure* script is located. If not specified conanfile.source_folder is used.

make(*target=None*, *args=None*)
Call the make program.

Parameters

- **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
- **args** – (Optional, Defaulted to `None`): List of arguments to use for the `make` call.

install(*args=None*, *target='install'*)
This is just an "alias" of `self.make(target="install")`

Parameters

- **args** – (Optional, Defaulted to `None`): List of arguments to use for the `make` call. By default an argument `DESTDIR=unix_path(self.package_folder)` is added to the call if the passed value is `None`. See more information about *tools.microsoft.unix_path() function*
- **target** – (Optional, Defaulted to `None`): Choose which target to install.

autoreconf(*args=None*)
Call `autoreconf`

Parameters args – (Optional, Defaulted to `None`): List of arguments to use for the `autoreconf` call.

A note about relocatable shared libraries in macOS built the Autotools build helper

When building a shared library with Autotools in macOS a section `LC_ID_DYLIB` and another `LC_LOAD_DYLIB` are added to the `.dylib`. These sections store `install_name` information, which is the location of the folder where the library or its dependencies are installed. You can check the install_name of your shared libraries using the otool command:

```
$ otool -l path/to/libMyLib.dylib
...
cmd LC_ID_DYLIB
    cmdsize 48
        name path/to/libMyLib.dylib (offset 24)
time stamp 1 Thu Jan  1 01:00:01 1970
    current version 1.0.0
compatibility version 1.0.0
...
Load command 11
        cmd LC_LOAD_DYLIB
    cmdsize 48
        name path/to/dependency.dylib (offset 24)
time stamp 2 Thu Jan  1 01:00:02 1970
    current version 1.0.0
```

(continues on next page)

(continued from previous page)

```
compatibility version 1.0.0
...
```

Why is this a problem when using Conan?

When using Conan the library will be built in the local cache and this means that this location will point to Conan's local cache folder where the library was installed. This location is where the library tells any other binaries using it where to load it at runtime. This is a problem since you can build the shared library in one machine, then upload it to a server and install it in another machine to use it. In this case, as Autotools behaves by default, you would have a library storing an `install_name` pointing to a folder that does not exist in your current machine so you would get linker errors when building.

How to adress this problem in Conan

The only thing Conan can do to make these shared libraries relocatable is to patch the built binaries after installation. To do this, when using the `Autotools` build helper and after running the Makefile's `install()` step, you can use the *fix_apple_shared_install_name()* tool to search for the built `.dylib` files and patch them by running the `install_name_tool` macOS utility, like this:

```
from conan.tools.apple import fix_apple_shared_install_name
class HelloConan(ConanFile):
  ...
  def package(self):
      autotools = Autotools(self)
      autotools.install()
      fix_apple_shared_install_name(self)
```

This will change the value of the `LC_ID_DYLIB` and `LC_LOAD_DYLIB` sections in the `.dylib` file to:

```
$ otool -l path/to/libMyLib.dylib
...
cmd LC_ID_DYLIB
    cmdsize 48
        name @rpath/libMyLib.dylib (offset 24)
time stamp 1 Thu Jan  1 01:00:01 1970
    current version 1.0.0
compatibility version 1.0.0
...
Load command 11
        cmd LC_LOAD_DYLIB
    cmdsize 48
        name @rpath/dependency.dylib (offset 24)
time stamp 2 Thu Jan  1 01:00:02 1970
    current version 1.0.0
compatibility version 1.0.0
```

The `@rpath` special keyword will tell the loader to search a list of paths to find the library. These paths can be defined by the consumer of that library by defining the `LC_RPATH` field. This is done by passing the `-Wl,-rpath -Wl,/path/to/libMyLib.dylib` linker flag when building the consumer of the library. Then if Conan builds an executable that consumes the `libMyLib.dylib` library, it will automatically add the `-Wl,-rpath -Wl,/path/to/libMyLib.dylib` flag so that the library is correctly found when building.

PkgConfigDeps

The `PkgConfigDeps` is the dependencies generator for pkg-config. Generates pkg-config files named `<PKG-NAME>.pc` containing a valid pkg-config file syntax.

This generator can be used by name in conanfiles:

Listing 23: conanfile.py

```
class Pkg(ConanFile):
    generators = "PkgConfigDeps"
```

Listing 24: conanfile.txt

```
[generators]
PkgConfigDeps
```

And it can also be fully instantiated in the conanfile `generate()` method:

```
from conan import ConanFile
from conan.tools.gnu import PkgConfigDeps

class App(ConanFile):
    settings = "os", "arch", "compiler", "build_type"
    requires = "zlib/1.2.11"

    def generate(self):
        pc = PkgConfigDeps(self)
        pc.generate()
```

Generated files

pkg-config format files named `<PKG-NAME>.pc`, containing a valid pkg-config file syntax. The `prefix` variable is automatically adjusted to the `package_folder`:

```
prefix=/Users/YOUR_USER/.conan/data/zlib/1.2.11/_/_/package/
↪647afeb69d3b0a2d3d316e80b24d38c714cc6900
libdir=${prefix}/lib
includedir=${prefix}/include

Name: zlib
Description: Conan package: zlib
Version: 1.2.11
Libs: -L"${libdir}" -lz -F Frameworks
Cflags: -I"${includedir}"
```

Customization

Naming

By default, the `*.pc` files will be named following these rules:

- For packages, it uses the package name, e.g., package `zlib/1.2.11` -> `zlib.pc`.

- For components, the package name + hyphen + component name, e.g., `openssl/3.0.0` with `self.cpp_info.components["crypto"]` -> `openssl-crypto.pc`.

You can change that default behavior with the `pkg_config_name` and `pkg_config_aliases` properties. See *Properties section below*.

If a recipe uses **components**, the files generated will be `<[PKG-NAME]-[COMP-NAME]>.pc` with their corresponding flags and require relations.

Additionally, a `<PKG-NAME>.pc` is generated to maintain compatibility for consumers with recipes that start supporting components. This `<PKG-NAME>.pc` file declares all the components of the package as requires while the rest of the fields will be empty, relying on the propagation of flags coming from the components `<[PKG-NAME]-[COMP-NAME]>.pc` files.

Reference

class PkgConfigDeps(*conanfile*)

content
Get all the .pc files content

generate()
Save all the **.pc* files

Attributes

build_context_activated

When you have a **build-require**, by default, the `*.pc` files are not generated. But you can activate it using the **build_context_activated** attribute:

```
tool_requires = ["my_tool/0.0.1"]
def generate(self):
    pc = PkgConfigDeps(self)
    # generate the *.pc file for the tool require
    pc.build_context_activated = ["my_tool"]
    pc.generate()
```

build_context_suffix

When you have the same package as a **build-require** and as a **regular require** it will cause a conflict in the generator because the file names of the `*.pc` files will collide as well as the names, requires names, etc.

For example, this is a typical situation with some requirements (capnproto, protobuf…) that contain a tool used to generate source code at build time (so it is a **build_require**), but also providing a library to link to the final application, so you also have a **regular require**. Solving this conflict is specially important when we are cross-building because the tool (that will run in the building machine) belongs to a different binary package than the library, that will “run” in the host machine.

You can use the **build_context_suffix** attribute to specify a suffix for a requirement, so the files/requires/names of the requirement in the build context (tool require) will be renamed:

```
tool_requires = ["my_tool/0.0.1"]
requires = ["my_tool/0.0.1"]
def generate(self):
    pc = PkgConfigDeps(self)
    # generate the *.pc file for the tool require
    pc.build_context_activated = ["my_tool"]
    # disambiguate the files, requires, names, etc
    pc.build_context_suffix = {"my_tool": "_BUILD"}
    pc.generate()
```

Properties

The following properties affect the PkgConfigDeps generator:

- **pkg_config_name** property will define the name of the generated *.pc file (xxxxx.pc)
- **pkg_config_aliases** property sets some aliases of any package/component name for *pkg_config* generator. This property only accepts list-like Python objects.
- **pkg_config_custom_content** property will add user defined content to the *.pc* files created by this generator.
- **component_version** property sets a custom version to be used in the Version field belonging to the created *.pc file for that component.

These properties can be defined at global cpp_info level or at component level.

Example:

```
def package_info(self):
    custom_content = "datadir=${prefix}/share"
    self.cpp_info.set_property("pkg_config_custom_content", custom_content)
    self.cpp_info.set_property("pkg_config_name", "myname")
    self.cpp_info.components["mycomponent"].set_property("pkg_config_name",
↪"componentname")
    self.cpp_info.components["mycomponent"].set_property("pkg_config_aliases", [
↪"alias1", "alias2"])
    self.cpp_info.components["mycomponent"].set_property("component_version", "1.14.12
↪")
```

PkgConfig

This tool can execute pkg_config executable to extract information from existing .pc files. This can be useful for example to create a "system" package recipe over some system installed library, as a way to automatically extract the .pc information from the system. Or if some proprietary package has a build system that only outputs .pc files.

Usage:

Read a pc file and access the information:

```
pkg_config = PkgConfig(conanfile, "libastral", pkg_config_path=<somedir>)

print(pkg_config.provides) # something like"libastral = 6.6.6"
print(pkg_config.version) # something like"6.6.6"
print(pkg_config.includedirs) # something like['/usr/local/include/libastral']
print(pkg_config.defines) # something like['_USE_LIBASTRAL']
print(pkg_config.libs) # something like['astral', 'm']
```

(continues on next page)

(continued from previous page)

```
print(pkg_config.libdirs) # something like['/usr/local/lib/libastral']
print(pkg_config.linkflags) # something like['-Wl,--whole-archive']
print(pkg_config.variables['prefix']) # something like'/usr/local'
```

Use the `pc` file information to fill a `cpp_info` object:

```
def package_info(self):
    pkg_config = PkgConfig(conanfile, "libastral", pkg_config_path=tmp_dir)
    pkg_config.fill_cpp_info(self.cpp_info, is_system=False, system_libs=["m", "rt"])
```

Reference

class PkgConfig(*conanfile*, *library*, *pkg_config_path=None*)

Parameters

- **conanfile** – The current recipe object. Always use `self`.
- **library** – The library which `.pc` file is to be parsed. It must exist in the pkg_config path.
- **pkg_config_path** – If defined it will be prepended to `PKG_CONFIG_PATH` environment variable, so the execution finds the required files.

fill_cpp_info(*cpp_info*, *is_system=True*, *system_libs=None*)

Method to fill a cpp_info object from the PkgConfig configuration

Parameters

- **cpp_info** – Can be the global one (self.cpp_info) or a component one (self.components["foo"].cpp_info).
- **is_system** – If `True`, all detected libraries will be assigned to `cpp_info.system_libs`, and none to `cpp_info.libs`.
- **system_libs** – If `True`, all detected libraries will be assigned to `cpp_info.system_libs`, and none to `cpp_info.libs`.

conf

This helper will listen to `tools.gnu:pkg_config` from the *global.conf* to define the `pkg_config` executable name or full path. It will by default it is `pkg-config`.

7.4.3 conan.tools.apple

XcodeDeps

The `XcodeDeps` tool is the dependency information generator for *Xcode*. It will generate multiple *.xcconfig* configuration files, the can be used by consumers using *xcodebuild* or *Xcode*. To use them just add the generated configuration files to the Xcode project or set the `-xcconfig` argument from the command line.

The `XcodeDeps` generator can be used by name in conanfiles:

Listing 25: conanfile.py

```
class Pkg(ConanFile):
    generators = "XcodeDeps"
```

Listing 26: conanfile.txt

```
[generators]
XcodeDeps
```

And it can also be fully instantiated in the conanfile `generate()` method:

Listing 27: conanfile.py

```
from conan import ConanFile
from conan.tools.apple import XcodeDeps

class Pkg(ConanFile):
    settings = "os", "compiler", "arch", "build_type"
    requires = "libpng/1.6.37@" # Note libpng has zlib as transitive dependency

    def generate(self):
        xcode = XcodeDeps(self)
        xcode.generate()
```

When the `XcodeDeps` generator is used, every invocation of `conan install` will generate several configuration files, per dependency and configuration. For the *conanfile.py* above, for example:

```
$ conan install conanfile.py # default is Release
$ conan install conanfile.py -s build_type=Debug
```

This generator is multi-configuration. It will generate different files for the different *Debug/Release* configurations for each requirement. It will also generate one single file (*conandeps.xcconfig*) aggregating all the files for the direct dependencies (just *libpng* in this case). The above commands generate the following files:

```
.
├── conan_config.xcconfig
├── conan_libpng.xcconfig
├── conan_libpng_libpng.xcconfig
├── conan_libpng_libpng_debug_x86_64.xcconfig
├── conan_libpng_libpng_release_x86_64.xcconfig
├── conan_zlib.xcconfig
├── conan_zlib_zlib.xcconfig
├── conan_zlib_zlib_debug_x86_64.xcconfig
├── conan_zlib_zlib_release_x86_64.xcconfig
└── conandeps.xcconfig
```

The first `conan install` with the default *Release* and *x86_64* configuration generates:

- *conan_libpng_libpng_release_x86_64.xcconfig*: declares variables with conditional logic to be considered only for the active configuration in *Xcode* or the one passed by command line to *xcodebuild*.
- *conan_libpng_libpng.xcconfig*: includes *conan_libpng_libpng_release_x86_64.xcconfig* and declares the following *Xcode* build settings: `HEADER_SEARCH_PATHS`, `GCC_PREPROCESSOR_DEFINITIONS`, `OTHER_CFLAGS`, `OTHER_CPLUSPLUSFLAGS`, `FRAMEWORK_SEARCH_PATHS`, `LIBRARY_SEARCH_PATHS`, `OTHER_LDFLAGS`. It also includes the generated *xcconfig* files for transitive dependencies (*conan_zlib_zlib.xcconfig* in this case).

- *conan_libpng.xcconfig*: in this case it only includes *conan_libpng_libpng.xcconfig*, but in the case that the required package has components, this file will include all of the components of the package.
- Same 3 files will be generated for each dependency in the graph. In this case, as *zlib* is a dependency of *libpng* it will generate: *conan_zlib_zlib_release_x86_64.xcconfig*, *conan_zlib_zlib.xcconfig* and *conan_zlib.xcconfig*.
- *conandeps.xcconfig*: configuration files including all direct dependencies, in this case, it just includes `conan_libpng.xcconfig`.
- The main *conan_config.xcconfig* file, to be added to the project. Includes both the files from this generator and the generated by the *XcodeToolchain* in case it was also set.

The second `conan install -s build_type=Debug` generates:

- *conan_libpng_libpng_debug_x86_64.xcconfig*: same variables as the one below for *Debug* configuration.
- *conan_libpng_libpng.xcconfig*: this file has been already created by the previous command, now it's modified to add the include for *conan_libpng_debug_x86_64.xcconfig*.
- *conan_libpng.xcconfig*: this file will remain the same.
- Like in the previous command the same 3 files will be generated for each dependency in the graph. In this case, as *zlib* is a dependency of *libpng* it will generate: *conan_zlib_zlib_debug_x86_64.xcconfig*, *conan_zlib_zlib.xcconfig* and *conan_zlib.xcconfig*.
- *conandeps.xcconfig*: configuration files including all direct dependencies, in this case, it just includes `conan_libpng.xcconfig`.
- The main *conan_config.xcconfig* file, to be added to the project. Includes both the files from this generator and the generated by the *XcodeToolchain* in case it was also set.

If you want to add this dependencies to you Xcode project, you just have to add the *conan_config.xcconfig* configuration file for all of the configurations you want to use (usually *Debug* and *Release*).

Additional variables defined

Besides the variables that define the *Xcode* build settings mentioned above, there are additional variables declared that may be useful to use in your *Xcode* project:

- *PACKAGE_ROOT_<package_name>*: Set to the location of the *package_folder attribute*.

Components support

This generator supports packages with components. That means that:

- If a **dependency** `package_info()` declares `cpp_info.requires` on some components, the generated *.xcconfig* files will contain includes to only those components.
- The current package `requires` will be fully dependent on and all components. Recall that the `package_info()` only applies for consumers, but not to the current package.

Custom configurations

If your Xcode project defines custom configurations, like `ReleaseShared`, or `MyCustomConfig`, it is possible to define it into the `XcodeDeps` generator, so different project configurations can use different set of dependencies. Let's say that our current project can be built as a shared library, with the custom configuration `ReleaseShared`, and the package also controls this with the `shared` option:

```
from conan import ConanFile
from conan.tools.apple import XcodeDeps

class Pkg(ConanFile):
    settings = "os", "compiler", "arch", "build_type"
    options = {"shared": [True, False]}
    default_options = {"shared": False}
    requires = "zlib/1.2.11"

    def generate(self):
        xcode = XcodeDeps(self)
        # We assume that -o *:shared=True is used to install all shared deps too
        if self.options.shared:
            xcode.configuration = str(self.settings.build_type) + "Shared"
        xcode.generate()
```

This will manage to generate new *.xcconfig* files for this custom configuration, and when you switch to this configuration in the IDE, the build system will take the correct values depending wether we want to link with shared or static libraries.

XcodeToolchain

The `XcodeToolchain` is the toolchain generator for Xcode. It will generate *.xcconfig* configuration files that can be added to Xcode projects. This generator translates the current package configuration, settings, and options, into Xcode *.xcconfig* files syntax.

The `XcodeToolchain` generator can be used by name in conanfiles:

Listing 28: conanfile.py

```
class Pkg(ConanFile):
    generators = "XcodeToolchain"
```

Listing 29: conanfile.txt

```
[generators]
XcodeToolchain
```

And it can also be fully instantiated in the conanfile `generate()` method:

```
from conan import ConanFile
from conan.tools.apple import XcodeToolchain

class App(ConanFile):
    settings = "os", "arch", "compiler", "build_type"

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

The `XcodeToolchain` will generate three files after a `conan install` command. As explained above for the XcodeDeps generator, each different configuration will create a set of files with different names. For example, running `conan install` for *Release* first and then *Debug* configuration:

```
$ conan install conanfile.py # default is Release
$ conan install conanfile.py -s build_type=Debug
```

Will create these files:

```
.
├── conan_config.xcconfig
├── conantoolchain_release_x86_64.xcconfig
├── conantoolchain_debug_x86_64.xcconfig
├── conantoolchain.xcconfig
└── conan_global_flags.xcconfig
```

Those files are:

- The main *conan_config.xcconfig* file, to be added to the project. Includes both the files from this generator and the generated by the *XcodeDeps* in case it was also set.
- *conantoolchain_<debug/release>_x86_64.xcconfig*: declares `CLANG_CXX_LIBRARY`, `CLANG_CXX_LANGUAGE_STANDARD` and `MACOSX_DEPLOYMENT_TARGET` variables with conditional logic depending on the build configuration, architecture and sdk set.
- *conantoolchain.xcconfig*: aggregates all the *conantoolchain_<config>_<arch>.xcconfig* files for the different installed configurations.
- *conan_global_flags.xcconfig*: this file will only be generated in case of any configuration variables related to compiler or linker flags are set. Check *the configuration section* below for more details.

Every invocation to `conan install` with different configuration will create a new *conantoolchain_<config>_<arch>.xcconfig* file that is aggregated in the *conantoolchain.xcconfig*, so you can have different configurations included in your Xcode project.

The XcodeToolchain files can declare the following Xcode build settings based on Conan settings values:

- `MACOSX_DEPLOYMENT_TARGET` is based on the value of the `os.version` setting and will make the build system to pass the flag `-mmacosx-version-min` with that value (if set). It defines the operating system version the binary should run into.
- `CLANG_CXX_LANGUAGE_STANDARD` is based on the value of the `compiler.cppstd` setting that sets the C++ language standard.
- `CLANG_CXX_LIBRARY` is based on the value of the compiler.libcxx setting and sets the version of the C++ standard library to use.

One of the advantages of using toolchains is that they can help to achieve the exact same build with local development flows, than when the package is created in the cache.

conf

This toolchain is also affected by these **[conf]** variables:

- `tools.build:cxxflags` list of C++ flags.
- `tools.build:cflags` list of pure C flags.
- `tools.build:sharedlinkflags` list of flags that will be used by the linker when creating a shared library.
- `tools.build:exelinkflags` list of flags that will be used by the linker when creating an executable.
- `tools.build:defines` list of preprocessor definitions.

If you set any of these variables, the toolchain will use them to generate the `conan_global_flags.xcconfig` file that will be included from the `conan_config.xcconfig` file.

XcodeBuild

The `XcodeBuild` build helper is a wrapper around the command line invocation of Xcode. It will abstract the calls like `xcodebuild -project app.xcodeproj -configuration <config> -arch <arch> ...`

The `XcodeBuild` helper can be used like:

```
from conan import conanfile
from conan.tools.apple import XcodeBuild

class App(ConanFile):
    settings = "os", "arch", "compiler", "build_type"

    def build(self):
        xcodebuild = XcodeBuild(self)
        xcodebuild.build("app.xcodeproj")
```

Reference

class XcodeBuild(*conanfile*)

__init__(*conanfile*)
Initialize self. See help(type(self)) for accurate signature.

`XcodeBuild.`**build**(*xcodeproj*, *target=None*)
Call to `xcodebuild` to build a Xcode project.

Parameters

- **xcodeproj** – the *xcodeproj* file to build.
- **target** – the target to build, in case this argument is passed to the `build()` method it will add the `-target` argument to the build system call. If not passed, it will build all the targets passing the `-alltargets` argument instead.

Returns the return code for the launched `xcodebuild` command.

The `Xcode.build()` method internally implements a call to `xcodebuild` like:

```
$ xcodebuild -project app.xcodeproj -configuration <configuration> -arch
↪<architecture> <sdk> <verbosity> -target <target>/-alltargets
```

Where:

- `configuration` is the configuration, typically *Release* or *Debug*, which will be obtained from `settings.build_type`.
- `architecture` is the build architecture, a mapping from the `settings.arch` to the common architectures defined by Apple 'i386', 'x86_64', 'armv7', 'arm64', etc.
- `sdk` is set based on the values of the `os.sdk` and `os.sdk_version` defining the `SDKROOT` Xcode build setting according to them. For example, setting `os.sdk=iOS` and *os.sdk_version=8.3*' will pass `SDKROOT=iOS8.3` to the build system. In case you defined the `tools.apple:sdk_path` in your **[conf]** this value will take preference and will directly pass `SDKROOT=<tools.apple:sdk_path>` so **take into account** that for this case the skd located in that path should set your `os.sdk` and `os.sdk_version` settings values.
- `verbosity` is the verbosity level for the build and can take value 'verbose' or 'quiet' if set by `tools.apple.xcodebuild:verbosity` in your **[conf]**

conf

- `tools.apple.xcodebuild:verbosity` verbosity value for the build, can be 'verbose' or 'quiet'
- `tools.apple:sdk_path` path for the sdk location, will set the `SDKROOT` value with preference over composing the value from the `os.sdk` and `os.sdk_version` settings.

conan.tools.apple.fix_apple_shared_install_name()

fix_apple_shared_install_name(*conanfile*)

Search for all the *dylib* files in the conanfile's *package_folder* and fix both the `LC_ID_DYLIB` and `LC_LOAD_DYLIB` fields on those files using the *install_name_tool* utility available in macOS to set `@rpath`.

This tool will search for all the *dylib* files in the conanfile's *package_folder* and fix the library *install names* (the `LC_ID_DYLIB` header). Libraries and executables inside the package folder will also have the `LC_LOAD_DYLIB` fields updated to reflect the patched install names. Executables inside the package will also get an `LC_RPATH` entry pointing to the relative location of the libraries inside the package folder. This is done using the *install_name_tool* utility available in macOS, as outlined below:

- For `LC_ID_DYLIB` which is the field containing the install name of the library, it will change the install name to one that uses the `@rpath`. For example, if the install name is `/path/to/lib/libname.dylib`, the new install name will be `@rpath/libname.dylib`. This is done by internally executing something like:

```
install_name_tool /path/to/lib/libname.dylib -id @rpath/libname.dylib
```

- For `LC_LOAD_DYLIB` which is the field containing the path to the library dependencies, it will change the path of the dependencies to one that uses the `@rpath`. For example, if a binary has a dependency on `/path/to/lib/dependency.dylib`, this will be updated to be `@rpath/dependency.dylib`. This is done for both libraries and executables inside the package folder, invoking *install_name_tool* as below:

```
install_name_tool /path/to/lib/libname.dylib -change /path/to/lib/dependency.dylib ↪@rpath/dependency.dylib
```

- For `LC_RPATH`, in those cases in which the packages also contain binary executables that depend on libraries within the same package, entries will be added to reflect the location of the libraries relative to the executable. If a package has executables in the *bin* subfolder and libraries in the *lib* subfolder, this can be performed with an invocation like this:

```
install_name_tool /path/to/bin/my_executable -add_rpath @executable_path/../lib
```

This tool is typically needed by recipes that use Autotools as the build system and in the case that the correct install names are not fixed in the library being packaged. Use this tool, if needed, in the conanfile's `package()` method like:

```
from conan.tools.apple import fix_apple_shared_install_name

class HelloConan(ConanFile):

  ...

  def package(self):
      autotools = Autotools(self)
      autotools.install()
      fix_apple_shared_install_name(self)
```

conan.tools.apple.is_apple_os()

is_apple_os(*conanfile*)
returns True if OS is Apple one (Macos, iOS, watchOS or tvOS

conan.tools.apple.to_apple_arch()

to_apple_arch(*conanfile*, *default=None*)
converts conan-style architecture into Apple-style arch

conan.tools.apple.XCRun()

class XCRun(*conanfile*, *sdk=None*, *use_settings_target=False*)
XCRun is a wrapper for the Apple **xcrun** tool used to get information for building.

Parameters

- **conanfile** – Conanfile instance.
- **sdk** – Will skip the flag when `False` is passed and will try to adjust the sdk it automatically if `None` is passed.
- **target_settings** – Try to use `settings_target` in case they exist (`False` by default)

find(*tool*)
find SDK tools (e.g. clang, ar, ranlib, lipo, codesign, etc.)

sdk_path
obtain sdk path (aka apple sysroot or -isysroot

sdk_version
obtain sdk version

sdk_platform_path
obtain sdk platform path

sdk_platform_version
obtain sdk platform version

cc
path to C compiler (CC)

cxx
path to C++ compiler (CXX)

ar
path to archiver (AR)

ranlib
path to archive indexer (RANLIB)

strip
path to symbol removal utility (STRIP)

libtool
path to libtool

7.4.4 conan.tools.env

Environment

`Environment` is a generic class that helps to define modifications to the environment variables. This class is used by other tools like the *conan.tools.gnu* autotools helpers and the *VirtualBuildEnv* and *VirtualRunEnv* generator. It is important to highlight that this is a generic class, to be able to use it, a specialization for the current context (shell script, bat file, path separators, etc), a `EnvVars` object needs to be obtained from it.

Variable declaration

```
from conan.tools.env import Environment

def generate(self):
    env = Environment()
    env.define("MYVAR1", "MyValue1")  # Overwrite previously existing MYVAR1 with new␣
↪value
    env.append("MYVAR2", "MyValue2")  # Append to existing MYVAR2 the new value
    env.prepend("MYVAR3", "MyValue3") # Prepend to existing MYVAR3 the new value
    env.remove("MYVAR3", "MyValue3")  # Remove the MyValue3 from MYVAR3
    env.unset("MYVAR4")               # Remove MYVAR4 definition from environment

    # And the equivalent with paths
    env.define_path("MYPATH1", "path/one")  # Overwrite previously existing MYPATH1␣
↪with new value
    env.append_path("MYPATH2", "path/two")  # Append to existing MYPATH2 the new value
    env.prepend_path("MYPATH3", "path/three") # Prepend to existing MYPATH3 the new␣
↪value
```

The "normal" variables (the ones declared with `define`, `append` and `prepend`) will be appended with a space, by default, but the `separator` argument can be provided to define a custom one.

The "path" variables (the ones declared with `define_path`, `append_path` and `prepend_path`) will be appended with the default system path separator, either `:` or `;`, but it also allows defining which one.

Composition

Environments can be composed:

```
from conan.tools.env import Environment

env1 = Environment()
env1.define(...)
env2 = Environment()
env2.append(...)

env1.compose_env(env2) # env1 has priority, and its modifications will prevail
```

Obtaining environment variables

You can obtain an `EnvVars` object with the `vars()` method like this:

```
from conan.tools.env import Environment

def generate(self):
    env = Environment()
    env.define("MYVAR1", "MyValue1")
    envvars = env.vars(self, scope="build")
    # use the envvars object
```

The default `scope` is equal `"build"`, which means that if this `envvars` generate a script to activate the variables, such script will be automatically added to the `conanbuild.sh|bat` one, for users and recipes convenience. Conan generators use `build` and `run` scope, but it might be possible to manage other scopes too.

Environment definition

There are some other places where `Environment` can be defined and used:

- In recipes `package_info()` method, in new `self.buildenv_info` and `self.runenv_info`, this environment will be propagated via `VirtualBuildEnv` and `VirtualRunEnv` respectively to packages depending on this recipe.
- In generators like `AutootoolsDeps`, `AutotoolsToolchain`, that need to define environment for the current recipe.
- In profiles `[buildenv]` section.
- In profiles `[runenv]` section.

The definition in `package_info()` is as follow, taking into account that both `self.buildenv_info` and `self.runenv_info` are objects of `Environment()` class.

```
from conan import ConanFile

class App(ConanFile):
    name = "mypkg"
    version = "1.0"
    settings = "os", "arch", "compiler", "build_type"

    def package_info(self):
        # This is information needed by consumers to build using this package
        self.buildenv_info.append("MYVAR", "MyValue")
        self.buildenv_info.prepend_path("MYPATH", "some/path/folder")

        # This is information needed by consumers to run apps that depends on this␣
↪package
        # at runtime
        self.runenv_info.define("MYPKG_DATA_DIR", os.path.join(self.package_folder,
                                                                "datadir"))
```

Reference

class Environment

Generic class that helps to define modifications to the environment variables.

dumps()

Returns A string with a profile-like original definition, not the full environment values

define(*name*, *value*, *separator=' '*)
Define *name* environment variable with value *value*

Parameters

- **name** – Name of the variable
- **value** – Value that the environment variable will take
- **separator** – The character to separate appended or prepended values

unset(*name*)
clears the variable, equivalent to a unset or set XXX=

Parameters **name** – Name of the variable to unset

append(*name*, *value*, *separator=None*)
Append the *value* to an environment variable *name*

Parameters

- **name** – Name of the variable to append a new value
- **value** – New value
- **separator** – The character to separate the appended value with the previous value. By default it will use a blank space.

append_path(*name*, *value*)
Similar to "append" method but indicating that the variable is a filesystem path. It will automatically handle the path separators depending on the operating system.

Parameters

- **name** – Name of the variable to append a new value
- **value** – New value

prepend(*name*, *value*, *separator=None*)
Prepend the *value* to an environment variable *name*

Parameters

- **name** – Name of the variable to prepend a new value
- **value** – New value
- **separator** – The character to separate the prepended value with the previous value

prepend_path(*name*, *value*)
Similar to "prepend" method but indicating that the variable is a filesystem path. It will automatically handle the path separators depending on the operating system.

Parameters

- **name** – Name of the variable to prepend a new value
- **value** – New value

remove(*name*, *value*)
Removes the *value* from the variable *name*.

Parameters

- **name** – Name of the variable
- **value** – Value to be removed.

compose_env (*other*)
Compose an Environment object with another one. `self` has precedence, the "other" will add/append if possible and not conflicting, but `self` mandates what to do. If `self` has `define()`, without placeholder, that will remain.

Parameters **other** (class:*Environment*) – the "other" Environment

vars (*conanfile*, *scope='build'*)
Return an EnvVars object from the current Environment object :param conanfile: Instance of a conanfile, usually `self` in a recipe :param scope: Determine the scope of the declared variables. :return:

deploy_base_folder (*package_folder*, *deploy_folder*)
Make the paths relative to the deploy_folder

EnvVars

`EnvVars` is a class that represents an instance of environment variables for a given system. It is obtained from the generic *Environment* class.

This class is used by other tools like the *conan.tools.gnu* autotools helpers and the *VirtualBuildEnv* and *VirtualRunEnv* generator.

Creating environment files

`EnvVars` object can generate environment files (shell, bat or powershell scripts):

```
def generate(self):
    env1 = Environment()
    env1.define("foo", "var")
    envvars = env1.vars(self)
    envvars.save_script("my_env_file")
```

Although it potentially could be used in other methods, this functionality is intended to work in the `generate()` method.

It will generate automatically a `my_env_file.bat` for Windows systems or `my_env_file.sh` otherwise.

In Windows, it is possible to opt-in to generate Powershell `.ps1` scripts instead of `.bat` ones, using the conf `tools.env.virtualenv:powershell=True`.

Also, by default, Conan will automatically append that launcher file path to a list that will be used to create a `conanbuild.bat|sh|ps1` file aggregating all the launchers in order. The `conanbuild.sh|bat|ps1` launcher will be created after the execution of the `generate()` method.

The `scope` argument (`"build"` by default) can be used to define different scope of environment files, to aggregate them separately. For example, using a `scope="run"`, like the `VirtualRunEnv` generator does, will aggregate and create a `conanrun.bat|sh|ps1` script:

```
def generate(self):
    env1 = Environment(self)
    env1.define("foo", "var")
    envvars = env1.vars(self, scope="run")
    # Will append "my_env_file" to "conanrun.bat|sh|ps1"
    envvars.save_script("my_env_file")
```

You can also use `scope=None` argument to avoid appending the script to the aggregated `conanbuild.bat|sh|ps1`:

```
env1 = Environment(self)
env1.define("foo", "var")
# Will not append "my_env_file" to "conanbuild.bat|sh|ps1"
envvars = env1.vars(self, scope=None)
envvars.save_script("my_env_file")
```

Running with environment files

The `conanbuild.bat|sh|ps1` launcher will be executed by default before calling every `self.run()` command. This would be typically done in the `build()` method.

You can change the default launcher with the `env` argument of `self.run()`:

```
...
def build(self):
    # This will automatically wrap the "foo" command with the correct environment:
    # source my_env_file.sh && foo
    # my_env_file.bat && foo
    # powershell my_env_file.ps1 ; cmd c/ foo
    self.run("foo", env=["my_env_file"])
```

Applying the environment variables

As an alternative to running a command, environments can be applied in the python environment:

```
from conan.tools.env import Environment

env1 = Environment(self)
env1.define("foo", "var")
envvars = env1.vars(self)
with envvars.apply():
   # Here os.getenv("foo") == "var"
   ...
```

Iterating the variables

You can iterate the environment variables of an `EnvVars` object like this:

```
env1 = Environment()
env1.append("foo", "var")
env1.append("foo", "var2")
envvars = env1.vars(self)
for name, value in envvars.items():
    assert name == "foo":
    assert value == "var var2"
```

The current value of the environment variable in the system is replaced in the returned value. This happens when variables are appended or prepended. If a placeholder is desired instead of the actual value, it is possible to use the `variable_reference` argument with a jinja template syntax, so a string with that resolved template will be returned instead:

```
env1 = Environment()
env1.append("foo", "var")
envvars = env1.vars(self)
for name, value in envvars.items(variable_reference="$penv{{{name}}}""):
    assert name == "foo":
    assert value == "$penv{{foo}} var"
```

Warning: In Windows, there is a limit to the size of environment variables, a total of 32K for the whole environment, but spcifically the PATH variable has a limit of 2048 characters. That means that the above utils could hit that limit, for example for large dependency graphs where all packages contribute to the PATH env-var.

This can be mitigated by:

- Putting the Conan cache closer to C:/ for shorter paths
- Better definition of what dependencies can contribute to the PATH env-var
- Other mechanisms for things like running with many shared libraries dependencies with too many .dlls, like `deployers`

Reference

class EnvVars(*conanfile*, *values*, *scope*)

Represents an instance of environment variables for a given system. It is obtained from the generic Environment class.

get(*name*, *default=None*, *variable_reference=None*)

get the value of a env-var

Parameters

- **name** – The name of the environment variable.
- **default** – The returned value if the variable doesn't exist, by default None.
- **variable_reference** – if specified, use a variable reference instead of the pre-existing value of environment variable, where {name} can be used to refer to the name of the variable.

items(*variable_reference=None*)

returns {str: str} (varname: value)

Parameters variable_reference – if specified, use a variable reference instead of the pre-existing value of environment variable, where {name} can be used to refer to the name of the variable.

apply()

Context manager to apply the declared variables to the current `os.environ` restoring the original environment when the context ends.

save_script(*filename*)

Saves a script file (bat, sh, ps1) with a launcher to set the environment. If the conf "tools.env.virtualenv:powershell" is set to True it will generate powershell launchers if Windows.

Parameters filename – Name of the file to generate. If the extension is provided, it will generate the launcher script for that extension, otherwise the format will be deduced checking if we are running inside Windows (checking also the subsystem) or not.

VirtualBuildEnv

VirtualBuildEnv is a generator that produces a *conanbuildenv* .bat or .sh script containing the environment variables of the build time context:

- From the `self.buildenv_info` of the direct `tool_requires` in "build" context.
- From the `self.runenv_info` of the transitive dependencies of those `tool_requires`.

It can be used by name in conanfiles:

Listing 30: conanfile.py

```
class Pkg(ConanFile):
    generators = "VirtualBuildEnv"
```

Listing 31: conanfile.txt

```
[generators]
VirtualBuildEnv
```

And it can also be fully instantiated in the conanfile `generate()` method:

Listing 32: conanfile.py

```
from conan import ConanFile
from conan.tools.env import VirtualBuildEnv

class Pkg(ConanFile):
    settings = "os", "compiler", "arch", "build_type"
    requires = "zlib/1.2.11", "bzip2/1.0.8"

    def generate(self):
        ms = VirtualBuildEnv(self)
        ms.generate()
```

Generated files

This generator (for example the invocation of `conan install --tool-require=cmake/3.20.0@ -g VirtualBuildEnv`) will create the following files:

- conanbuildenv-release-x86_64.(bat|sh): This file contains the actual definition of environment variables like PATH, LD_LIBRARY_PATH, etc, and any other variable defined in the dependencies `buildenv_info` corresponding to the `build` context, and to the current installed configuration. If a repeated call is done with other settings, a different file will be created. After the execution or sourcing of this file, a new deactivation script will be generated, capturing the current environment, so the environment can be restored when desired. The file will be named also following the current active configuration, like `deactivate_conanbuildenv-release-x86_64.bat`.
- conanbuild.(bat|sh): Accumulates the calls to one or more other scripts, in case there are multiple tools in the generate process that create files, to give one single convenient file for all. This only calls the latest specific configuration one, that is, if `conan install` is called first for Release build type, and then for Debug, `conanbuild.(bat|sh)` script will call the Debug one.
- deactivate_conanbuild.(bat|sh): Accumulates the deactivation calls defined in the above `conanbuild.(bat|sh)`. This file should only be called after the accumulated activate has been called first.

Reference

class VirtualBuildEnv(*conanfile*)

Calculates the environment variables of the build time context and produces a conanbuildenv .bat or .sh script

environment()

Returns an `Environment` object containing the environment variables of the build context.

Returns an `Environment` object instance containing the obtained variables.

vars(*scope='build'*)

Parameters **scope** – Scope to be used.

Returns An `EnvVars` instance containing the computed environment variables.

generate(*scope='build'*)

Produces the launcher scripts activating the variables for the build context.

Parameters **scope** – Scope to be used.

VirtualRunEnv

`VirtualRunEnv` is a generator that produces a launcher *conanrunenv* .bat or .sh script containing environment variables of the run time environment.

The launcher contains the runtime environment information, anything that is necessary in the environment to actually run the compiled executables and applications. The information is obtained from:

- The `self.runenv_info` of the dependencies corresponding to the `host` context.
- Also automatically deduced from the `self.cpp_info` definition of the package, to define `PATH`, `LD_LIBRARY_PATH`, `DYLD_LIBRARY_PATH` and `DYLD_FRAMEWORK_PATH` environment variables.

It can be used by name in conanfiles:

Listing 33: conanfile.py

```
class Pkg(ConanFile):
    generators = "VirtualRunEnv"
```

Listing 34: conanfile.txt

```
[generators]
VirtualRunEnv
```

And it can also be fully instantiated in the conanfile `generate()` method:

Listing 35: conanfile.py

```
from conan import ConanFile
from conan.tools.env import VirtualRunEnv

class Pkg(ConanFile):
    settings = "os", "compiler", "arch", "build_type"
    requires = "zlib/1.2.11", "bzip2/1.0.8"

    def generate(self):
        ms = VirtualRunEnv(self)
        ms.generate()
```

Generated files

- conanrunenv-release-x86_64.(bat|sh): This file contains the actual definition of environment variables like PATH, LD_LIBRARY_PATH, etc, and `runenv_info` of dependencies corresponding to the `host` context, and to the current installed configuration. If a repeated call is done with other settings, a different file will be created.
- conanrun.(bat|sh): Accumulates the calls to one or more other scripts to give one single convenient file for all. This only calls the latest specific configuration one, that is, if `conan install` is called first for Release build type, and then for Debug, `conanrun.(bat|sh)` script will call the Debug one.

After the execution of one of those files, a new deactivation script will be generated, capturing the current environment, so the environment can be restored when desired. The file will be named also following the current active configuration, like `deactivate_conanrunenv-release-x86_64.bat`.

Reference

class VirtualRunEnv(*conanfile*)

Calculates the environment variables of the runtime context and produces a conanrunenv .bat or .sh script

Parameters **conanfile** – The current recipe object. Always use `self`.

environment()

Returns an `Environment` object containing the environment variables of the run context.

Returns an `Environment` object instance containing the obtained variables.

vars(*scope='run'*)

Parameters **scope** – Scope to be used.

Returns An `EnvVars` instance containing the computed environment variables.

generate(*scope='run'*)

Produces the launcher scripts activating the variables for the run context.

Parameters **scope** – Scope to be used.

7.4.5 conan.tools.build

Building

conan.tools.build.build_jobs()

build_jobs(*conanfile*)

Returns the number of CPUs available for parallel builds. It returns the configuration value for `tools.build:jobs` if exists, otherwise, it defaults to the helper function `_cpu_count()`. `_cpu_count()` reads cgroup to detect the configured number of CPUs. Currently, there are two versions of cgroup available.

In the case of cgroup v1, if the data in cgroup is invalid, processor detection comes into play. Whenever processor detection is not enabled, `build_jobs()` will safely return 1.

In the case of cgroup v2, if no limit is set, processor detection is used. When the limit is set, the behavior is as described in cgroup v1.

Parameters **conanfile** – The current recipe object. Always use `self`.

Returns `int` with the number of jobs

conan.tools.build.cross_building()

cross_building(*conanfile=None*, *skip_x64_x86=False*)

Check if we are cross building comparing the *build* and *host* settings. Returns `True` in the case that we are cross-building.

Parameters

- **conanfile** – The current recipe object. Always use `self`.
- **skip_x64_x86** – Do not consider cross building when building to 32 bits from 64 bits: x86_64 to x86, sparcv9 to sparc or ppc64 to ppc32

Returns `True` if we are cross building, `False` otherwise.

conan.tools.build.can_run()

can_run(*conanfile*)

Validates whether is possible to run a non-native app on the same architecture. It's an useful feature for the case your architecture can run more than one target. For instance, Mac M1 machines can run both *armv8* and *x86_64*.

Parameters **conanfile** – The current recipe object. Always use `self`.

Returns `bool` value from `tools.build.cross_building:can_run` if exists, otherwise, it returns `False` if we are cross-building, else, `True`.

Cppstd

conan.tools.build.check_min_cppstd()

check_min_cppstd(*conanfile*, *cppstd*, *gnu_extensions=False*)

Check if current cppstd fits the minimal version required.

In case the current cppstd doesn't fit the minimal version required by cppstd, a ConanInvalidConfiguration exception will be raised.

1. If settings.compiler.cppstd, the tool will use settings.compiler.cppstd to compare
2. It not settings.compiler.cppstd, the tool will use compiler to compare (reading the default from cppstd_default)
3. If not settings.compiler is present (not declared in settings) will raise because it cannot compare.
4. If can not detect the default cppstd for settings.compiler, a exception will be raised.

Parameters

- **conanfile** – The current recipe object. Always use `self`.
- **cppstd** – Minimal cppstd version required
- **gnu_extensions** – GNU extension is required (e.g gnu17)

conan.tools.build.check_max_cppstd()

check_max_cppstd(*conanfile*, *cppstd*, *gnu_extensions=False*)

Check if current cppstd fits the maximum version required.

In case the current cppstd doesn't fit the maximum version required by cppstd, a ConanInvalidConfiguration exception will be raised.

1. If settings.compiler.cppstd, the tool will use settings.compiler.cppstd to compare
2. It not settings.compiler.cppstd, the tool will use compiler to compare (reading the default from cppstd_default)
3. If not settings.compiler is present (not declared in settings) will raise because it cannot compare.
4. If can not detect the default cppstd for settings.compiler, a exception will be raised.

Parameters

- **conanfile** – The current recipe object. Always use `self`.
- **cppstd** – Maximum cppstd version required
- **gnu_extensions** – GNU extension is required (e.g gnu17)

conan.tools.build.valid_min_cppstd()

valid_min_cppstd(*conanfile*, *cppstd*, *gnu_extensions=False*)

Validate if current cppstd fits the minimal version required.

Parameters

- **conanfile** – The current recipe object. Always use `self`.
- **cppstd** – Minimal cppstd version required
- **gnu_extensions** – GNU extension is required (e.g gnu17). This option ONLY works on Linux.

Returns True, if current cppstd matches the required cppstd version. Otherwise, False.

conan.tools.build.valid_max_cppstd()

valid_max_cppstd(*conanfile*, *cppstd*, *gnu_extensions=False*)

Validate if current cppstd fits the maximum version required.

Parameters

- **conanfile** – The current recipe object. Always use `self`.
- **cppstd** – Maximum cppstd version required
- **gnu_extensions** – GNU extension is required (e.g gnu17). This option ONLY works on Linux.

Returns True, if current cppstd matches the required cppstd version. Otherwise, False.

conan.tools.build.default_cppstd()

default_cppstd(*conanfile*, *compiler=None*, *compiler_version=None*)
Get the default `compiler.cppstd` for the "conanfile.settings.compiler" and "conanfile settings.compiler_version" or for the parameters "compiler" and "compiler_version" if specified.

Parameters

- **conanfile** – The current recipe object. Always use `self`.
- **compiler** – Name of the compiler e.g. gcc
- **compiler_version** – Version of the compiler e.g. 12

Returns The default `compiler.cppstd` for the specified compiler

conan.tools.build.supported_cppstd()

supported_cppstd(*conanfile*, *compiler=None*, *compiler_version=None*)
Get the a list of supported `compiler.cppstd` for the "conanfile.settings.compiler" and "conanfile.settings.compiler_version" or for the parameters "compiler" and "compiler_version" if specified.

Parameters

- **conanfile** – The current recipe object. Always use `self`.
- **compiler** – Name of the compiler e.g: gcc
- **compiler_version** – Version of the compiler e.g: 12

Returns a list of supported `cppstd` values.

7.4.6 conan.tools.files

conan.tools.files basic operations

conan.tools.files.copy()

copy(*conanfile*, *pattern*, *src*, *dst*, *keep_path=True*, *excludes=None*, *ignore_case=True*)
Copy the files matching the pattern (fnmatch) at the src folder to a dst folder.

Parameters

- **conanfile** – The current recipe object. Always use `self`.
- **pattern** – (Required) An fnmatch file pattern of the files that should be copied. It must not start with `..` relative path or an exception will be raised.
- **src** – (Required) Source folder in which those files will be searched. This folder will be stripped from the dst parameter. E.g., lib/Debug/x86.
- **dst** – (Required) Destination local folder. It must be different from src value or an exception will be raised.
- **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.
- **excludes** – (Optional, defaulted to `None`) A tuple/list of fnmatch patterns or even a single one to be excluded from the copy.

- **ignore_case** – (Optional, defaulted to `True`) If enabled, it will do a case-insensitive pattern matching. will do a case-insensitive pattern matching when `True`

Returns list of copied files

Usage:

```
def package(self):
    copy(self, "*.h", self.source_folder, os.path.join(self.package_folder, "include
↪"))
    copy(self, "*.lib", self.build_folder, os.path.join(self.package_folder, "lib"))
```

Note: The files that are **symlinks to files** or **symlinks to folders** with be treated like any other file, so they will only be copied if the specified pattern matches with the file.

At the destination folder, the symlinks will be created pointing to the exact same file or folder, absolute or relative, being the responsibility of the user to manipulate the symlink to, for example, transform the symlink into a relative path before copying it so it points to the destination folder.

Check *here* the reference of tools to manage symlinks.

conan.tools.files.load()

load(*conanfile*, *path*, *encoding='utf-8'*)

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.

Parameters

- **conanfile** – The current recipe object. Always use `self`.
- **path** – Path to the file to read
- **encoding** – (Optional, Defaulted to `utf-8`): Specifies the input file text encoding.

Returns The contents of the file

Usage:

```
from conan.tools.files import load

content = load(self, "myfile.txt")
```

conan.tools.files.save()

save(*conanfile*, *path*, *content*, *append=False*, *encoding='utf-8'*)

Utility function to save files in one line. It will manage the open and close of the file and creating directories if necessary.

Parameters

- **conanfile** – The current recipe object. Always use `self`.
- **path** – Path of the file to be created.
- **content** – Content (str or bytes) to be write to the file.

- **append** – (Optional, Defaulted to False): If `True` the contents will be appended to the existing one.
- **encoding** – (Optional, Defaulted to utf-8): Specifies the output file text encoding.

Usage:

```
from conan.tools.files import save

save(self, "path/to/otherfile.txt", "contents of the file")
```

conan.tools.files.rename()

rename(*conanfile*, *src*, *dst*)

Utility functions to rename a file or folder src to dst with retrying. `os.rename()` frequently raises "Access is denied" exception on Windows. This function renames file or folder using robocopy to avoid the exception on Windows.

Parameters

- **conanfile** – The current recipe object. Always use `self`.
- **src** – Path to be renamed.
- **dst** – Path to be renamed to.

Usage:

```
from conan.tools.files import rename

def source(self):
    rename(self, "lib-sources-abe2h9fe", "sources")  # renaming a folder
```

conan.tools.files.replace_in_file()

replace_in_file(*conanfile*, *file_path*, *search*, *replace*, *strict=True*, *encoding='utf-8'*)

Replace a string `search` in the contents of the file `file_path` with the string replace.

Parameters

- **conanfile** – The current recipe object. Always use `self`.
- **file_path** – File path of the file to perform the replacing.
- **search** – String you want to be replaced.
- **replace** – 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.
- **encoding** – (Optional, Defaulted to utf-8): Specifies the input and output files text encoding.

Usage:

```
from conan.tools.files import replace_in_file

replace_in_file(self, os.path.join(self.source_folder, "folder", "file.txt"), "foo",
↪"bar")
```

conan.tools.files.rm()

rm(*conanfile*, *pattern*, *folder*, *recursive=False*)

Utility functions to remove files matching a `pattern` in a `folder`.

Parameters

- **conanfile** – The current recipe object. Always use `self`.
- **pattern** – Pattern that the files to be removed have to match (fnmatch).
- **folder** – Folder to search/remove the files.
- **recursive** – If `recursive` is specified it will search in the subfolders.

Usage:

```
from conan.tools.files import rm

rm(self, "*.tmp", self.build_folder, recursive=True)
```

conan.tools.files.mkdir()

mkdir(*conanfile*, *path*)

Utility functions to create a directory. The existence of the specified directory is checked, so mkdir() will do nothing if the directory already exists.

Parameters

- **conanfile** – The current recipe object. Always use `self`.
- **path** – Path to the folder to be created.

Usage:

```
from conan.tools.files import mkdir

mkdir(self, "mydir") # Creates mydir if it does not already exist
mkdir(self, "mydir") # Does nothing
```

conan.tools.files.rmdir()

rmdir(*conanfile*, *path*)

Usage:

```
from conan.tools.files import rmdir

rmdir(self, "mydir") # Remove mydir if it exist
rmdir(self, "mydir") # Does nothing
```

conan.tools.files.chdir()

chdir(*conanfile*, *newdir*)

This is a context manager that allows to temporary change the current directory in your conanfile

Parameters

- **conanfile** – The current recipe object. Always use `self`.
- **newdir** – Directory path name to change the current directory.

Usage:

```
from conan.tools.files import chdir

def build(self):
    with chdir(self, "./subdir"):
        do_something()
```

conan.tools.files.unzip()

This function extract different compressed formats (`.tar.gz`, `.tar`, `.tzb2`, `.tar.bz2`, `.tgz`, `.txz`, `tar.xz`, and `.zip`) into the given destination folder.

It also accepts gzipped files, with extension `.gz` (not matching any of the above), and it will unzip them into a file with the same name but without the extension, or to a filename defined by the `destination` argument.

```
from conan.tools.files import unzip

unzip(self, "myfile.zip")
# or to extract in "myfolder" sub-folder
unzip(self, "myfile.zip", "myfolder")
```

You can keep the permissions of the files using the `keep_permissions=True` parameter.

```
from conan.tools.files import unzip

unzip(self, "myfile.zip", "myfolder", keep_permissions=True)
```

Use the `pattern` argument if you want to filter specific files and paths to decompress from the archive.

```
from conan.tools.files import unzip

# Extract only files inside relative folder "small"
unzip(self, "bigfile.zip", pattern="small/*")
# Extract only txt files
unzip(self, "bigfile.zip", pattern="*.txt")
```

unzip(*conanfile*, *filename*, *destination='.'*, *keep_permissions=False*, *pattern=None*, *strip_root=False*)
Extract different compressed formats

Parameters

- **conanfile** – The current recipe object. Always use `self`.
- **filename** – Path to the compressed file.
- **destination** – (Optional, Defaulted to `.`) Destination folder (or file for .gz files)
- **keep_permissions** – (Optional, Defaulted to `False`) Keep the zip permissions. WARNING: Can be dangerous if the zip was not created in a NIX system, the bits could produce undefined permission schema. Use this option only if you are sure that the zip was created correctly.
- **pattern** – (Optional, Defaulted to `None`) Extract only paths matching the pattern. This should be a Unix shell-style wildcard, see fnmatch documentation for more details.

- **strip_root** – (Optional, Defaulted to False) If True, and all the unzipped contents are in a single folder it will flat the folder moving all the contents to the parent folder.

conan.tools.files.update_conandata()

This function reads the `conandata.yml` inside the exported folder in the conan cache, if it exists. If the `conandata.yml` does not exist, it will create it. Then, it updates the conandata dictionary with the provided `data` one, which is updated recursively, prioritizing the `data` values, but keeping other existing ones. Finally the `conandata.yml` is saved in the same place.

This helper can only be used within the `export()` method, it can raise otherwise. One application is to capture in the `conandata.yml` the scm coordinates (like Git remote url and commit), to be able to recover it later in the `source()` method and have reproducible recipes that can build from sources without actually storing the sources in the recipe.

Usage:

```
from conan import ConanFile
from conan.tools.files import update_conandata

class Pkg(ConanFile):
    name = "pkg"
    version = "0.1"

    def export(self):
        # This is an example, doesn't make sense to have static data, instead you
        # could put the data directly in a conandata.yml file.
        # This would be useful for storing dynamic data, obtained at export() time␣
↪from elsewhere
        update_conandata(self, {"mydata": {"value": {"nested1": 123, "nested2": "some-
↪string"}}})

    def source(self):
        data = self.conan_data["sources"]["mydata"]
```

update_conandata (*conanfile*, *data*)

Tool to modify the `conandata.yml` once it is exported. It can be used, for example:

- To add additional data like the “commit” and “url” for the scm.
- To modify the contents cleaning the data that belong to other versions (different from the exported) to avoid changing the recipe revision when the changed data doesn’t belong to the current version.

Parameters

- **conanfile** – The current recipe object. Always use `self`.
- **data** – (Required) A dictionary (can be nested), of values to update

conan.tools.files.collect_libs()

collect_libs (*conanfile*, *folder=None*)

Returns a sorted list of library names from the libraries (files with extensions *.so*, *.lib*, *.a* and *.dylib*) located inside the `conanfile.cpp_info.libdirs` (by default) or the **folder** directory relative to the package folder. Useful to collect not inter-dependent libraries or with complex names like `libmylib-x86-debug-en.lib`.

For UNIX libraries staring with **lib**, like *libmath.a*, this tool will collect the library name **math**.

Parameters

- **conanfile** – The current recipe object. Always use `self`.
- **folder** – (Optional, Defaulted to `None`): String indicating the subfolder name inside `conanfile.package_folder` where the library files are.

Returns A list with the library names

> **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.

Usage:

```
from conan.tools.files import collect_libs

def package_info(self):
    self.cpp_info.libdirs = ["lib", "other_libdir"]  # Default value is 'lib'
    self.cpp_info.libs = collect_libs(self)
```

For UNIX libraries starting with **lib**, like *libmath.a*, this tool will collect the library name **math**. Regarding symlinks, this tool will keep only the "most generic" file among the resolved real file and all symlinks pointing to this real file. For example among files below, this tool will select *libmath.dylib* file and therefore only append *math* in the returned list:

```
-rwxr-xr-x libmath.1.0.0.dylib lrwxr-xr-x libmath.1.dylib -> libmath.1.0.0.dylib
lrwxr-xr-x libmath.dylib -> libmath.1.dylib
```

conan.tools.files downloads

conan.tools.files.get()

get (*conanfile*, *url*, *md5=None*, *sha1=None*, *sha256=None*, *destination='.'*, *filename=''*, *keep_permissions=False*, *pattern=None*, *verify=True*, *retry=None*, *retry_wait=None*, *auth=None*, *headers=None*, *strip_root=False*)

High level download and decompressing of a tgz, zip or other compressed format file. 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`.

Parameters

- **conanfile** – The current recipe object. Always use `self`.
- **destination** – (Optional defaulted to `.`) Destination folder
- **filename** – (Optional defaulted to '') If provided, the saved file will have the specified name, otherwise it is deduced from the URL
- **url** – forwarded to `tools.file.download()`.
- **md5** – forwarded to `tools.file.download()`.
- **sha1** – forwarded to `tools.file.download()`.
- **sha256** – forwarded to `tools.file.download()`.

- **keep_permissions** – forwarded to `tools.file.unzip()`.
- **pattern** – forwarded to `tools.file.unzip()`.
- **verify** – forwarded to `tools.file.download()`.
- **retry** – forwarded to `tools.file.download()`.
- **retry_wait** – S forwarded to `tools.file.download()`.
- **auth** – forwarded to `tools.file.download()`.
- **headers** – forwarded to `tools.file.download()`.
- **strip_root** – forwarded to `tools.file.unzip()`.

conan.tools.files.ftp_download()

ftp_download(*conanfile*, *host*, *filename*, *login=''*, *password=''*)

Ftp download of a file. Retrieves a file from an FTP server. This doesn't support SSL, but you might implement it yourself using the standard Python FTP library.

Parameters

- **conanfile** – The current recipe object. Always use `self`.
- **host** – IP or host of the FTP server
- **filename** – Path to the file to be downloaded
- **login** – Authentication login
- **password** – Authentication password

Usage:

```
from conan.tools.files import ftp_download

def source(self):
    ftp_download(self, 'ftp.debian.org', "debian/README")
    self.output.info(load("README"))
```

conan.tools.files.download()

download(*conanfile*, *url*, *filename*, *verify=True*, *retry=None*, *retry_wait=None*, *auth=None*, *headers=None*, *md5=None*, *sha1=None*, *sha256=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.

You can pass hash checking parameters: `md5`, `sha1`, `sha256`. All the specified algorithms will be checked. If any of them doesn't match, the downloaded file will be removed and it will raise a `ConanException`.

Parameters

- **conanfile** – The current recipe object. Always use `self`.
- **url** – URL to download. It can be a list, which only the first one will be downloaded, and the follow URLs will be used as mirror in case of download error. Files accessible in the local filesystem can be referenced with a URL starting with `file:///` followed by an absolute path to a file (where the third `/` implies `localhost`).
- **filename** – Name of the file to be created in the local storage

- **verify** – When False, disables https certificate validation
- **retry** – Number of retries in case of failure. Default is overridden by "tools.files.download:retry" conf
- **retry_wait** – Seconds to wait between download attempts. Default is overriden by "tools.files.download:retry_wait" conf.
- **auth** – A tuple of user and password to use HTTPBasic authentication
- **headers** – A dictionary with additional headers
- **md5** – MD5 hash code to check the downloaded file
- **sha1** – SHA-1 hash code to check the downloaded file
- **sha256** – SHA-256 hash code to check the downloaded file

Usage:

```
download(self, "http://someurl/somefile.zip", "myfilename.zip")

# to disable verification:
download(self, "http://someurl/somefile.zip", "myfilename.zip", verify=False)

# to retry the download 2 times waiting 5 seconds between them
download(self, "http://someurl/somefile.zip", "myfilename.zip", retry=2, retry_wait=5)

# Use https basic authentication
download(self, "http://someurl/somefile.zip", "myfilename.zip", auth=("user",
↪"password"))

# Pass some header
download(self, "http://someurl/somefile.zip", "myfilename.zip", headers={"Myheader":
↪"My value"})

# Download and check file checksum
download(self, "http://someurl/somefile.zip", "myfilename.zip", md5=
↪"e5d695597e9fa520209d1b41edad2a27")

# to add mirrors
download(self, ["https://ftp.gnu.org/gnu/gcc/gcc-9.3.0/gcc-9.3.0.tar.gz",
                "http://mirror.linux-ia64.org/gnu/gcc/releases/gcc-9.3.0/gcc-9.3.0.
↪tar.gz"],
                "gcc-9.3.0.tar.gz",
               sha256=
↪"5258a9b6afe9463c2e56b9e8355b1a4bee125ca828b8078f910303bc2ef91fa6")
```

conf

It uses these *configuration entries*:

- `tools.files.download:retry`: number of retries in case some error occurs.
- `tools.files.download:retry_wait`: seconds to wait between retries.

conan.tools.files patches

conan.tools.files.patch()

patch(*conanfile*, *base_path=None*, *patch_file=None*, *patch_string=None*, *strip=0*, *fuzz=False*, ***kwargs*)

Applies a diff from file (patch_file) or string (patch_string) in the conanfile.source_folder directory. The folder containing the sources can be customized with the self.folders attribute in the layout(self) method.

Parameters

- **base_path** – The path is a relative path to conanfile.export_sources_folder unless an absolute path is provided.
- **patch_file** – Patch file that should be applied. The path is relative to the conanfile.source_folder unless an absolute path is provided.
- **patch_string** – Patch string that should be applied.
- **strip** – Number of folders to be stripped from the path.
- **output** – Stream object.
- **fuzz** – Should accept fuzzy patches.
- **kwargs** – Extra parameters that can be added and will contribute to output information

Usage:

```
from conan.tools.files import patch

def build(self):
    for it in self.conan_data.get("patches", {}).get(self.version, []):
        patch(self, **it)
```

conan.tools.files.apply_conandata_patches()

apply_conandata_patches(*conanfile*)

Applies patches stored in `conanfile.conan_data` (read from `conandata.yml` file). It will apply all the patches under `patches` entry that matches the given `conanfile.version`. If versions are not defined in `conandata.yml` it will apply all the patches directly under `patches` keyword.

The key entries will be passed as kwargs to the `patch` function.

Usage:

```
from conan.tools.files import apply_conandata_patches

def build(self):
    apply_conandata_patches(self)
```

Examples of `conandata.yml`:

```
patches:
- patch_file: "patches/0001-buildflatbuffers-cmake.patch"
- patch_file: "patches/0002-implicit-copy-constructor.patch"
  base_path: "subfolder"
  patch_type: backport
  patch_source: https://github.com/google/flatbuffers/pull/5650
  patch_description: Needed to build with modern clang compilers.
```

With different patches for different versions:

```yaml
patches:
  "1.11.0":
    - patch_file: "patches/0001-buildflatbuffers-cmake.patch"
    - patch_file: "patches/0002-implicit-copy-constructor.patch"
      base_path: "subfolder"
      patch_type: backport
      patch_source: https://github.com/google/flatbuffers/pull/5650
      patch_description: Needed to build with modern clang compilers.
  "1.12.0":
    - patch_file: "patches/0001-buildflatbuffers-cmake.patch"
    - patch_string: |
        --- a/tests/misc-test.c
        +++ b/tests/misc-test.c
        @@ -1232,6 +1292,8 @@ main (int argc, char **argv)
              g_test_add_func ("/misc/pause-cancel", do_pause_cancel_test);
              g_test_add_data_func ("/misc/stealing/async", GINT_TO_POINTER (FALSE),␣
↪do_stealing_test);
              g_test_add_data_func ("/misc/stealing/sync", GINT_TO_POINTER (TRUE), do_
↪stealing_test);
        +     g_test_add_func ("/misc/response/informational/content-length", do_
↪response_informational_content_length_test);
        +

        ret = g_test_run ();
    - patch_file: "patches/0003-fix-content-length-calculation.patch"
```

conan.tools.files.export_conandata_patches()

export_conandata_patches (*conanfile*)

Exports patches stored in 'conanfile.conan_data' (read from 'conandata.yml' file). It will export all the patches under 'patches' entry that matches the given 'conanfile.version'. If versions are not defined in 'conandata.yml' it will export all the patches directly under 'patches' keyword.

Example of `conandata.yml` without versions defined:

```python
from conan.tools.files import export_conandata_patches
def export_sources(self):
    export_conandata_patches(self)
```

conan.tools.files checksums

conan.tools.files.check_md5()

check_md5 (*conanfile*, *file_path*, *signature*)

Check that the specified `md5sum` of the `file_path` matches with `signature`. If doesn't match it will raise a `ConanException`.

Parameters

- **conanfile** – The current recipe object. Always use `self`.
- **file_path** – Path of the file to check.
- **signature** – Expected md5sum.

conan.tools.files.check_sha1()

check_sha1(*conanfile*, *file_path*, *signature*)

Check that the specified `sha1` of the `file_path` matches with signature. If doesn't match it will raise a `ConanException`.

Parameters

- **conanfile** – Conanfile object.
- **file_path** – Path of the file to check.
- **signature** – Expected sha1sum

conan.tools.files.check_sha256()

check_sha256(*conanfile*, *file_path*, *signature*)

Check that the specified `sha256` of the `file_path` matches with signature. If doesn't match it will raise a `ConanException`.

Parameters

- **conanfile** – Conanfile object.
- **file_path** – Path of the file to check.
- **signature** – Expected sha256sum

conan.tools.files.symlinks

conan.tools.files.symlinks.absolute_to_relative_symlinks()

absolute_to_relative_symlinks(*conanfile*, *base_folder*)

Convert the symlinks with absolute paths into relative ones if they are pointing to a file or directory inside the `base_folder`. Any absolute symlink pointing outside the `base_folder` will be ignored.

Parameters

- **conanfile** – The current recipe object. Always use `self`.
- **base_folder** – Folder to be scanned.

conan.tools.files.symlinks.remove_external_symlinks()

remove_external_symlinks(*conanfile*, *base_folder*)

Remove the symlinks to files that point outside the `base_folder`, no matter if relative or absolute.

Parameters

- **conanfile** – The current recipe object. Always use `self`.
- **base_folder** – Folder to be scanned.

conan.tools.files.symlinks.remove_broken_symlinks()

remove_broken_symlinks(*conanfile*, *base_folder=None*)

Remove the broken symlinks, no matter if relative or absolute.

Parameters

- **conanfile** – The current recipe object. Always use `self`.
- **base_folder** – Folder to be scanned.

conan.tools.files AutoPackager

The `AutoPackager` together with the layout() feature, allow to automatically package the files following the declared information in the `layout()` method:

It will copy:

- Files from `self.cpp.local.includedirs` to `self.cpp.package.includedirs`
- Files from `self.cpp.local.libdirs` to `self.cpp.package.libdirs`
- Files from `self.cpp.local.bindirs` to `self.cpp.package.bindirs`
- Files from `self.cpp.local.srcdirs` to `self.cpp.package.srcdirs`
- Files from `self.cpp.local.builddirs` to `self.cpp.package.builddirs`
- Files from `self.cpp.local.resdirs` to `self.cpp.package.resdirs`
- Files from `self.cpp.local.frameworkdirs` to `self.cpp.package.frameworkdirs`

The patterns of the files to be copied can be defined with the *.patterns* property of the `AutoPackager` instance. The default patterns are:

```
packager = AutoPackager(self)
packager.patterns.include == ["*.h", "*.hpp", "*.hxx"]
packager.patterns.lib == ["*.so", "*.so.*", "*.a", "*.lib", "*.dylib"]
packager.patterns.bin == ["*.exe", "*.dll"]
packager.patterns.src == []
packager.patterns.build == []
packager.patterns.res == []
packager.patterns.framework == []
```

Usage:

```
from conan import ConanFile
from conan.tools.files import AutoPackager

class Pkg(ConanFile):

    def layout(self):
        ...

    def package(self):
        packager = AutoPackager(self)
        packager.patterns.include = ["*.hpp", "*.h", "include3.h"]
        packager.patterns.lib = ["*.a"]
        packager.patterns.bin = ["*.exe"]
        packager.patterns.src = ["*.cpp"]
```

(continues on next page)

(continued from previous page)

```
        packager.patterns.framework = ["sframe*", "bframe*"]
        packager.run()
```

class AutoPackager(*conanfile*)

Parameters **conanfile** – The current recipe object. Always use `self`.

7.4.7 conan.tools.meson

MesonToolchain

Important: This class will generate files that are only compatible with Meson versions >= 0.55.0

The `MesonToolchain` is the toolchain generator for Meson and it can be used in the `generate()` method as follows:

```
from conan import ConanFile
from conan.tools.meson import MesonToolchain

class App(ConanFile):
    settings = "os", "arch", "compiler", "build_type"
    requires = "hello/0.1"
    options = {"shared": [True, False]}
    default_options = {"shared": False}

    def generate(self):
        tc = MesonToolchain(self)
        tc.preprocessor_definitions["MYDEFINE"] = "MYDEF_VALUE"
        tc.generate()
```

Important: When your recipe has dependencies `MesonToolchain` only works with the `PkgConfigDeps` generator. Please, do not use other generators, as they can have overlapping definitions that can conflict.

Generated files

The `MesonToolchain` generates the following files after a **conan install** (or when building the package in the cache) with the information provided in the `generate()` method as well as information translated from the current `settings`, `conf`, etc.:

- *conan_meson_native.ini*: if doing a native build.
- *conan_meson_cross.ini*: if doing a cross-build (*conan.tools.build*).

conan_meson_native.ini

This file contains the definitions of all the Meson properties related to the Conan options and settings for the current package, platform, etc. This includes but is not limited to the following:

- Detection of `default_library` from Conan settings.

  - Based on existence/value of an option named `shared`.
- Detection of `buildtype` from Conan settings.
- Definition of the C++ standard as necessary.
- The Visual Studio runtime (`b_vscrt`), obtained from Conan input settings.

conan_meson_cross.ini

This file contains the same information as the previous *conan_meson_native.ini*, but with additional information to describe host, target, and build machines (such as the processor architecture).

Check out the meson documentation for more details on native and cross files:

- Machine files
- Native environments
- Cross compilation

Default directories

`MesonToolchain` manages some of the directories used by Meson. These are variables declared under the `[project options]` section of the files *conan_meson_native.ini* and *conan_meson_cross.ini* (see more information about Meson directories):

`bindir`: value coming from `self.cpp.package.bindirs`. Defaulted to None. `sbindir`: value coming from `self.cpp.package.bindirs`. Defaulted to None. `libexecdir`: value coming from `self.cpp.package.bindirs`. Defaulted to None. `datadir`: value coming from `self.cpp.package.resdirs`. Defaulted to None. `localedir`: value coming from `self.cpp.package.resdirs`. Defaulted to None. `mandir`: value coming from `self.cpp.package.resdirs`. Defaulted to None. `infodir`: value coming from `self.cpp.package.resdirs`. Defaulted to None. `includedir`: value coming from `self.cpp.package.includedirs`. Defaulted to None. `libdir`: value coming from `self.cpp.package.libdirs`. Defaulted to None.

Notice that it needs a `layout` to be able to initialize those `self.cpp.package.xxxxx` variables. For instance:

```
from conan import ConanFile
from conan.tools.meson import MesonToolchain
class App(ConanFile):
    settings = "os", "arch", "compiler", "build_type"
    def layout(self):
        self.folders.build = "build"
        self.cpp.package.resdirs = ["res"]
    def generate(self):
        tc = MesonToolchain(self)
        self.output.info(tc.project_options["datadir"])  # Will print '["res"]'
        tc.generate()
```

Note: All of them are saved only if they have any value. If the values are``None``, they won't be mentioned in `[project options]` section.

Customization

Attributes

definitions

This attribute allows defining Meson project options:

```
def generate(self):
    tc = MesonToolchain(self)
    tc.definitions["MYVAR"] = "MyValue"
    tc.generate()
```

This is translated to:

- One project options definition for `MYVAR` in `conan_meson_native.ini` or `conan_meson_cross.ini` file.

preprocessor_definitions

This attribute allows defining compiler preprocessor definitions, for multiple configurations (Debug, Release, etc).

```
def generate(self):
    tc = MesonToolchain(self)
    tc.preprocessor_definitions["MYDEF"] = "MyValue"
    tc.generate()
```

This is translated to:

- One preprocessor definition for `MYDEF` in `conan_meson_native.ini` or `conan_meson_cross.ini` file.

conf

`MesonToolchain` is affected by these `[conf]` variables:

- `tools.meson.mesontoolchain:backend`. the meson backend to use. Possible values: `ninja`, `vs`, `vs2010`, `vs2015`, `vs2017`, `vs2019`, `xcode`.
- `tools.apple:sdk_path` argument for SDK path in case of Apple cross-compilation. It is used as value of the flag `-isysroot`.
- `tools.android:ndk_path` argument for NDK path in case of Android cross-compilation. It is used to get some binaries like `c`, `cpp` and `ar` used in `[binaries]` section from *conan_meson_cross.ini*.
- `tools.build:cxxflags` list of extra C++ flags that is used by `cpp_args`.
- `tools.build:cflags` list of extra of pure C flags that is used by `c_args`.
- `tools.build:sharedlinkflags` list of extra linker flags that is used by `c_link_args` and `cpp_link_args`.
- `tools.build:exelinkflags` list of extra linker flags that is used by `c_link_args` and `cpp_link_args`.

- `tools.build:linker_scripts` list of linker scripts, each of which will be prefixed with `-T` and passed to `c_link_args` and `cpp_link_args`. Only use this flag with linkers that supports specifying linker scripts with the `-T` flag, such as `ld`, `gold`, and `lld`.
- `tools.build:compiler_executables` dict-like Python object which specifies the compiler as key and the compiler executable path as value. Those keys will be mapped as follows:
  - `c`: will set `c` in `[binaries]` section from *conan_meson_xxxx.ini*.
  - `cpp`: will set `cpp` in `[binaries]` section from *conan_meson_xxxx.ini*.
  - `objc`: will set `objc` in `[binaries]` section from *conan_meson_xxxx.ini*.
  - `objcpp`: will set `objcpp` in `[binaries]` section from *conan_meson_xxxx.ini*.

Cross-building for Apple and Android

The `MesonToolchain` adds all the flags required to cross-compile for Apple (MacOS M1, iOS, etc.) and Android.

Apple

It adds link flags `-arch XXX`, `-isysroot [SDK_PATH]` and the minimum deployment target flag, e.g., `-mios-version-min=8.0` to the `MesonToolchain` `c_args`, `c_link_args`, `cpp_args`, and `cpp_link_args` attributes, given the Conan settings for any Apple OS (iOS, watchOS, etc.) and the `tools.apple:sdk_path` configuration value like it's shown in this example of host profile:

Listing 36: **ios_host_profile**

```
[settings]
os = iOS
os.version = 10.0
os.sdk = iphoneos
arch = armv8
compiler = apple-clang
compiler.version = 12.0
compiler.libcxx = libc++

[conf]
tools.apple:sdk_path=/my/path/to/iPhoneOS.sdk
```

Objective-C arguments

In Apple OS's there are also specific Objective-C/Objective-C++ arguments: `objc`, `objcpp`, `objc_args`, `objc_link_args`, `objcpp_args`, and `objcpp_link_args`, as public attributes of the `MesonToolchain` class, where the variables `objc` and `objcpp` are initialized as `clang` and `clang++` respectively by default.

Android

It initializes the `MesonToolchain` `c`, `cpp`, and `ar` attributes, which are needed to cross-compile for Android, given the Conan settings for Android and the `tools.android:ndk_path` configuration value like it's shown in this example of host profile:

Listing 37: **android_host_profile**

```
[settings]
os = Android
os.api_level = 21
```

(continues on next page)

(continued from previous page)

```
arch = armv8

[conf]
tools.android:ndk_path=/my/path/to/NDK
```

Read more

- *Getting started with Meson*

Reference

class MesonToolchain(*conanfile*, *backend=None*)

MesonToolchain generator

Parameters

- **conanfile** – `< ConanFile object >` The current recipe object. Always use `self`.
- **backend** – `str` `backend` Meson variable value. By default, `ninja`.

properties = None

Dict-like object that defines Meson‘‘properties‘‘ with `key=value` format

project_options = None

Dict-like object that defines Meson `project options` with `key=value` format

preprocessor_definitions = None

Dict-like object that defines Meson `preprocessor definitions`

pkg_config_path = None

Defines the Meson `pkg_config_path` variable

cross_build = None

Dict-like object with the build, host, and target as the Meson machine context

c = None

Defines the Meson `c` variable. Defaulted to `CC` build environment value

cpp = None

Defines the Meson `cpp` variable. Defaulted to `CXX` build environment value

c_ld = None

Defines the Meson `c_ld` variable. Defaulted to `CC_LD` or `LD` build environment value

cpp_ld = None

Defines the Meson `cpp_ld` variable. Defaulted to `CXX_LD` or `LD` build environment value

ar = None

Defines the Meson `ar` variable. Defaulted to `AR` build environment value

strip = None

Defines the Meson `strip` variable. Defaulted to `STRIP` build environment value

as_ = None

Defines the Meson `as` variable. Defaulted to `AS` build environment value

windres = None

Defines the Meson `windres` variable. Defaulted to `WINDRES` build environment value

pkgconfig = None
Defines the Meson `pkgconfig` variable. Defaulted to `PKG_CONFIG` build environment value

c_args = None
Defines the Meson `c_args` variable. Defaulted to `CFLAGS` build environment value

c_link_args = None
Defines the Meson `c_link_args` variable. Defaulted to `LDFLAGS` build environment value

cpp_args = None
Defines the Meson `cpp_args` variable. Defaulted to `CXXFLAGS` build environment value

cpp_link_args = None
Defines the Meson `cpp_link_args` variable. Defaulted to `LDFLAGS` build environment value

apple_arch_flag = None
Apple arch flag as a list, e.g., `["-arch", "i386"]`

apple_isysroot_flag = None
Apple sysroot flag as a list, e.g., `["-isysroot", "./Platforms/MacOSX.platform"]`

apple_min_version_flag = None
Apple minimum binary version flag as a list, e.g., `["-mios-version-min", "10.8"]`

objc = None
Defines the Meson `objc` variable. Defaulted to `None`, if if any Apple OS `clang`

objcpp = None
Defines the Meson `objcpp` variable. Defaulted to `None`, if if any Apple OS `clang++`

objc_args = None
Defines the Meson `objc_args` variable. Defaulted to `OBJCFLAGS` build environment value

objc_link_args = None
Defines the Meson `objc_link_args` variable. Defaulted to `LDFLAGS` build environment value

objcpp_args = None
Defines the Meson `objcpp_args` variable. Defaulted to `OBJCXXFLAGS` build environment value

objcpp_link_args = None
Defines the Meson `objcpp_link_args` variable. Defaulted to `LDFLAGS` build environment value

generate()
Creates a `conan_meson_native.ini` (if native builds) or a `conan_meson_cross.ini` (if cross builds) with the proper content. If Windows OS, it will be created a `conanvcvars.bat` as well.

Meson

The `Meson()` build helper is intended to be used in the `build()` and `package()` methods, to call Meson commands automatically.

```
from conan import ConanFile
from conan.tools.meson import Meson

class PkgConan(ConanFile):

    def build(self):
        meson = Meson(self)
        meson.configure()
        meson.build()
```

(continues on next page)

(continued from previous page)

```
    def package(self):
        meson = Meson(self)
        meson.install()
```

Reference

class Meson(*conanfile*)

This class calls Meson commands when a package is being built. Notice that this one should be used together with the `MesonToolchain` generator.

Parameters **conanfile** – < `ConanFile object` > The current recipe object. Always use `self`.

configure(*reconfigure=False*)

Runs `meson setup [FILE] "BUILD_FOLDER" "SOURCE_FOLDER" [-Dprefix=PACKAGE_FOLDER]` command, where `FILE` could be `--native-file conan_meson_native.ini` (if native builds) or `--cross-file conan_meson_cross.ini` (if cross builds).

Parameters **reconfigure** – `bool` value that adds `--reconfigure` param to the final command.

build(*target=None*)

Runs `meson compile -C . -j[N_JOBS] [TARGET]` in the build folder. You can specify `N_JOBS` through the configuration line `tools.build:jobs=N_JOBS` in your profile `[conf]` section.

Parameters **target** – `str` Specifies the target to be executed.

install()

Runs `meson install -C "."` in the build folder. Notice that it will execute `self.configure(reconfigure=True)` at first.

test()

Runs `meson test -v -C "."` in the build folder.

conf

The `Meson` build helper is affected by these `[conf]` variables:

- `tools.meson.mesontoolchain:extra_machine_files=[<FILENAME>]` configuration to add your machine files at the end of the command using the correct parameter depending on native or cross builds. See this Meson reference for more information.

7.4.8 conan.tools.system

conan.tools.system.package_manager

The tools under *conan.tools.system.package_manager* are wrappers around some of the most popular system package managers for different platforms. You can use them to invoke system package managers in recipes and perform the most typical operations, like installing a package, updating the package manager database or checking if a package is installed. By default, when you invoke them they will not try to install anything on the system, to change this behavior you can set the value of the `tools.system.package_manager:mode` *configuration*.

You can use these tools inside the `system_requirements()` method of your recipe, like:

Listing 38: conanfile.py

```
from conan.tools.system.package_manager import Apt, Yum, PacMan, Zypper

def system_requirements(self):
    # depending on the platform or the tools.system.package_manager:tool configuration
    # only one of these will be executed
    Apt(self).install(["libgl-dev"])
    Yum(self).install(["libglvnd-devel"])
    PacMan(self).install(["libglvnd"])
    Zypper(self).install(["Mesa-libGL-devel"])
```

Conan will automatically choose which package manager to use by looking at the Operating System name. In the example above, if we are running on Ubuntu Linux, Conan will ignore all the calls except for the `Apt()` one and will only try to install the packages using the `apt-get` tool. Conan uses the following mapping by default:

- *Apt* for **Linux** with distribution names: *ubuntu*, *debian* or *raspbian*
- *Yum* for **Linux** with distribution names: *pidora*, *scientific*, *xenserver*, *amazon*, *oracle*, *amzn*, *almalinux* or *rocky*
- *Dnf* for **Linux** with distribution names: *fedora*, *rhel*, *centos*, *mageia*
- *Brew* for **macOS**
- *PacMan* for **Linux** with distribution names: *arch*, *manjaro* and when using **Windows** with *msys2*
- *Chocolatey* for **Windows**
- *Zypper* for **Linux** with distribution names: *opensuse*, *sles*
- *Pkg* for **Linux** with distribution names: *freebsd*
- *PkgUtil* for **Solaris**

You can override this default mapping and set the package manager tool you want to use by default setting the configuration property *tools.system.package_manager:tool*.

Methods available for system package manager tools

All these wrappers share three methods that represent the most common operations with a system package manager. They take the same form for all of the package managers except for *Apt* that also accepts the *recommends* argument for the *install method*.

- `install(self, packages, update=False, check=False)`: try to install the list of packages passed as a parameter. If the parameter `check` is `True` it will check if those packages are already installed before installing them. If the parameter `update` is `True` it will try to update the package manager database before checking and installing. Its behaviour is affected by the value of `tools.system.package_manager:mode` *configuration*. It will return the return code of the executed commands.
- `install_substitutes(packages_substitutes, update=False, check=True)`: try to install the list of lists of substitutes packages passed as a parameter, e.g., `[["pkg1", "pkg2"], ["pkg3"]]`. It succeeds if one of the substitutes list is completely installed, so it's intended to be used when you have different packages for different distros. Internally, it's calling the previous `install(packages, update=update, check=check)` method, so `update` and `check` have the same purpose as above.
- `update()` update the system package manager database. Its behaviour is affected by the value of `tools.system.package_manager:mode` *configuration*.

- `check(packages)` check if the list of packages passed as parameter are already installed. It will return a list with the packages that are missing.

Configuration properties that affect how system package managers are invoked

As explained above there are several `[conf]` that affect how these tools are invoked:

- `tools.system.package_manager:tool`: to choose which package manager tool you want to use by default: `"apt-get"`, `"yum"`, `"dnf"`, `"brew"`, `"pacman"`, `"choco"`, `"zypper"`, `"pkg"` or `"pkgutil"`
- `tools.system.package_manager:mode`: mode to use when invoking the package manager tool. There are two possible values:
  - `"check"`: it will just check for missing packages at most and will not try to update the package manager database or install any packages in any case. It will raise an error if required packages are not installed in the system. This is the default value.
  - `"report"`: Just capture the `.install()` calls to capture packages, but do not check nor install them. Never raises an error. Mostly useful for `conan graph info` commands.
  - `"report-installed"`: Report, without failing which packages are needed (same as `report`) and also check which of them are actually installed in the current system.
  - `"install"`: it will allow Conan to perform update or install operations.
- `tools.system.package_manager:sudo`: Use *sudo* when invoking the package manager tools in Linux (`False` by default)
- `tools.system.package_manager:sudo_askpass`: Use the `-A` argument if using sudo in Linux to invoke the system package manager (`False` by default)

There are some specific arguments for each of these tools. Here is the complete reference:

conan.tools.system.package_manager.Apt

Will invoke the *apt-get* command. Enabled by default for **Linux** with distribution names: *ubuntu* and *debian*.

Reference

class Apt(*conanfile*, *arch_names=None*)

Parameters

- **conanfile** – The current recipe object. Always use `self`.
- **arch_names** – This argument maps the Conan architecture setting with the package manager tool architecture names. It is `None` by default, which means that it will use a default mapping for the most common architectures. For example, if you are using `x86_64` Conan architecture setting, it will map this value to `amd64` for *Apt* and try to install the `<package_name>:amd64` package.

install(*packages*, *update=False*, *check=False*, *recommends=False*)

Will try to install the list of packages passed as a parameter. Its behaviour is affected by the value of `tools.system.package_manager:mode` *configuration*.

Parameters

- **packages** – try to install the list of packages passed as a parameter.
- **update** – try to update the package manager database before checking and installing.
- **check** – check if the packages are already installed before installing them.
- **recommends** – if the parameter `recommends` is `False` it will add the `'--no-install-recommends'` argument to the *apt-get* command call.

Returns the return code of the executed apt command.

check(**args*, ***kwargs*)
Check if the list of packages passed as parameter are already installed.

Parameters **packages** – list of packages to check.

Returns list of packages from the packages argument that are not installed in the system.

install_substitutes(**args*, ***kwargs*)
Will try to call the install() method with several lists of packages passed as a variable number of parameters. This is useful if, for example, the names of the packages are different from one distro or distro version to another. For example, `libxcb` for `Apt` is named `libxcb-util-dev` in Ubuntu >= 15.0 and `libxcb-util0-dev` for other versions. You can call to:

```
# will install the first list of packages that succeeds in the␣
↪installation
Apt.install_substitutes(["libxcb-util-dev"], ["libxcb-util0-dev"])
```

Parameters

- **packages_alternatives** – try to install the list of packages passed as a parameter.
- **update** – try to update the package manager database before checking and installing.
- **check** – check if the packages are already installed before installing them.

Returns the return code of the executed package manager command.

update(**args*, ***kwargs*)
Update the system package manager database. Its behaviour is affected by the value of `tools.system.package_manager:mode` *configuration*.

Returns the return code of the executed package manager update command.

You can pass the `arch_names` argument to override the default Conan mapping like this:

Listing 39: conanfile.py

```
...
def system_requirements(self):
    apt = Apt(self, arch_names={"<conan_arch_setting>": "apt_arch_setting"})
    apt.install(["libgl-dev"])
```

The default mapping that Conan uses for *APT* packages architecture is:

```
self._arch_names = {"x86_64": "x86_64",
                    "x86": "i?86",
                    "ppc32": "powerpc",
                    "ppc64le": "ppc64le",
                    "armv7": "armv7",
                    "armv7hf": "armv7hl",
```

(continues on next page)

(continued from previous page)

```
                    "armv8": "aarch64",
                    "s390x": "s390x"} if arch_names is None else arch_names
```

conan.tools.system.package_manager.Yum

Will invoke the *yum* command. Enabled by default for **Linux** with distribution names: *pidora*, *scientific*, *xenserver*, *amazon*, *oracle*, *amzn* and *almalinux*.

Reference

class Yum(*conanfile*, *arch_names=None*)

Parameters

- **conanfile** – the current recipe object. Always use `self`.
- **arch_names** – this argument maps the Conan architecture setting with the package manager tool architecture names. It is `None` by default, which means that it will use a default mapping for the most common architectures. For example, if you are using `x86` Conan architecture setting, it will map this value to `i?86` for *Yum* and try to install the `<package_name>.i?86` package.

check(**args*, ***kwargs*)

Check if the list of packages passed as parameter are already installed.

Parameters **packages** – list of packages to check.

Returns list of packages from the packages argument that are not installed in the system.

install(**args*, ***kwargs*)

Will try to install the list of packages passed as a parameter. Its behaviour is affected by the value of `tools.system.package_manager:mode` *configuration*.

Parameters

- **packages** – try to install the list of packages passed as a parameter.
- **update** – try to update the package manager database before checking and installing.
- **check** – check if the packages are already installed before installing them.

Returns the return code of the executed package manager command.

install_substitutes(**args*, ***kwargs*)

Will try to call the install() method with several lists of packages passed as a variable number of parameters. This is useful if, for example, the names of the packages are different from one distro or distro version to another. For example, `libxcb` for `Apt` is named `libxcb-util-dev` in Ubuntu >= 15.0 and `libxcb-util0-dev` for other versions. You can call to:

```
# will install the first list of packages that succeeds in the␣
↪installation
Apt.install_substitutes(["libxcb-util-dev"], ["libxcb-util0-dev"])
```

Parameters

- **packages_alternatives** – try to install the list of packages passed as a parameter.
- **update** – try to update the package manager database before checking and installing.

- **check** – check if the packages are already installed before installing them.

Returns the return code of the executed package manager command.

update (**args*, ***kwargs*)

Update the system package manager database. Its behaviour is affected by the value of `tools.system.package_manager:mode` *configuration*.

Returns the return code of the executed package manager update command.

The default mapping Conan uses for *Yum* packages architecture is:

```
self._arch_names = {"x86_64": "x86_64",
                    "x86": "i?86",
                    "ppc32": "powerpc",
                    "ppc64le": "ppc64le",
                    "armv7": "armv7",
                    "armv7hf": "armv7hl",
                    "armv8": "aarch64",
                    "s390x": "s390x"} if arch_names is None else arch_names
```

conan.tools.system.package_manager.Dnf

Will invoke the *dnf* command. Enabled by default for **Linux** with distribution names: *fedora*, *rhel*, *centos* and *mageia*. This tool has exactly the same default values, constructor and methods than the *Yum* tool.

conan.tools.system.package_manager.PacMan

Will invoke the *pacman* command. Enabled by default for **Linux** with distribution names: *arch*, *manjaro* and when using **Windows** with *msys2*

Reference

class PacMan (*conanfile*, *arch_names=None*)

Parameters

- **conanfile** – the current recipe object. Always use `self`.
- **arch_names** – this argument maps the Conan architecture setting with the package manager tool architecture names. It is `None` by default, which means that it will use a default mapping for the most common architectures. If you are using `x86` Conan architecture setting, it will map this value to `lib32` for *PacMan* and try to install the `<package_name>-lib32` package.

check (**args*, ***kwargs*)

Check if the list of packages passed as parameter are already installed.

Parameters **packages** – list of packages to check.

Returns list of packages from the packages argument that are not installed in the system.

install (**args*, ***kwargs*)

Will try to install the list of packages passed as a parameter. Its behaviour is affected by the value of `tools.system.package_manager:mode` *configuration*.

Parameters

- **packages** – try to install the list of packages passed as a parameter.
- **update** – try to update the package manager database before checking and installing.
- **check** – check if the packages are already installed before installing them.

Returns the return code of the executed package manager command.

install_substitutes (**args*, ***kwargs*)
Will try to call the install() method with several lists of packages passed as a variable number of parameters. This is useful if, for example, the names of the packages are different from one distro or distro version to another. For example, `libxcb` for `Apt` is named `libxcb-util-dev` in Ubuntu >= 15.0 and `libxcb-util0-dev` for other versions. You can call to:

```
# will install the first list of packages that succeeds in the␣
↪installation
Apt.install_substitutes(["libxcb-util-dev"], ["libxcb-util0-dev"])
```

Parameters

- **packages_alternatives** – try to install the list of packages passed as a parameter.
- **update** – try to update the package manager database before checking and installing.
- **check** – check if the packages are already installed before installing them.

Returns the return code of the executed package manager command.

update (**args*, ***kwargs*)
Update the system package manager database. Its behaviour is affected by the value of `tools.system.package_manager:mode` *configuration*.

Returns the return code of the executed package manager update command.

The default mapping Conan uses for *PacMan* packages architecture is:

```
self._arch_names = {"x86": "lib32"} if arch_names is None else arch_names
```

conan.tools.system.package_manager.Zypper

Will invoke the *zypper* command. Enabled by default for **Linux** with distribution names: *opensuse*, *sles*.

Reference

class Zypper (*conanfile*)

Parameters conanfile – The current recipe object. Always use `self`.

check (**args*, ***kwargs*)
Check if the list of packages passed as parameter are already installed.

Parameters packages – list of packages to check.

Returns list of packages from the packages argument that are not installed in the system.

install (**args*, ***kwargs*)
Will try to install the list of packages passed as a parameter. Its behaviour is affected by the value of `tools.system.package_manager:mode` *configuration*.

Parameters

- **packages** – try to install the list of packages passed as a parameter.
- **update** – try to update the package manager database before checking and installing.
- **check** – check if the packages are already installed before installing them.

Returns the return code of the executed package manager command.

install_substitutes(**args*, ***kwargs*)

Will try to call the install() method with several lists of packages passed as a variable number of parameters. This is useful if, for example, the names of the packages are different from one distro or distro version to another. For example, `libxcb` for `Apt` is named `libxcb-util-dev` in Ubuntu >= 15.0 and `libxcb-util0-dev` for other versions. You can call to:

```
# will install the first list of packages that succeeds in the␣
↪installation
Apt.install_substitutes(["libxcb-util-dev"], ["libxcb-util0-dev"])
```

Parameters

- **packages_alternatives** – try to install the list of packages passed as a parameter.
- **update** – try to update the package manager database before checking and installing.
- **check** – check if the packages are already installed before installing them.

Returns the return code of the executed package manager command.

update(**args*, ***kwargs*)

Update the system package manager database. Its behaviour is affected by the value of `tools.system.package_manager:mode` *configuration*.

Returns the return code of the executed package manager update command.

conan.tools.system.package_manager.Brew

Will invoke the *brew* command. Enabled by default for **macOS**.

Reference

class Brew(*conanfile*)

Parameters conanfile – The current recipe object. Always use `self`.

check(**args*, ***kwargs*)

Check if the list of packages passed as parameter are already installed.

Parameters packages – list of packages to check.

Returns list of packages from the packages argument that are not installed in the system.

install(**args*, ***kwargs*)

Will try to install the list of packages passed as a parameter. Its behaviour is affected by the value of `tools.system.package_manager:mode` *configuration*.

Parameters

- **packages** – try to install the list of packages passed as a parameter.
- **update** – try to update the package manager database before checking and installing.

- **check** – check if the packages are already installed before installing them.

Returns the return code of the executed package manager command.

install_substitutes (**args*, ***kwargs*)
Will try to call the install() method with several lists of packages passed as a variable number of parameters. This is useful if, for example, the names of the packages are different from one distro or distro version to another. For example, `libxcb` for `Apt` is named `libxcb-util-dev` in Ubuntu >= 15.0 and `libxcb-util0-dev` for other versions. You can call to:

```
# will install the first list of packages that succeeds in the␣
↪installation
Apt.install_substitutes(["libxcb-util-dev"], ["libxcb-util0-dev"])
```

Parameters

- **packages_alternatives** – try to install the list of packages passed as a parameter.
- **update** – try to update the package manager database before checking and installing.
- **check** – check if the packages are already installed before installing them.

Returns the return code of the executed package manager command.

update (**args*, ***kwargs*)
Update the system package manager database. Its behaviour is affected by the value of `tools.system.package_manager:mode` *configuration*.

Returns the return code of the executed package manager update command.

conan.tools.system.package_manager.Pkg

Will invoke the *pkg* command. Enabled by default for **Linux** with distribution names: *freebsd*.

Reference

class Pkg (*conanfile*)

Parameters **conanfile** – The current recipe object. Always use `self`.

check (**args*, ***kwargs*)
Check if the list of packages passed as parameter are already installed.

Parameters **packages** – list of packages to check.

Returns list of packages from the packages argument that are not installed in the system.

install (**args*, ***kwargs*)
Will try to install the list of packages passed as a parameter. Its behaviour is affected by the value of `tools.system.package_manager:mode` *configuration*.

Parameters

- **packages** – try to install the list of packages passed as a parameter.
- **update** – try to update the package manager database before checking and installing.
- **check** – check if the packages are already installed before installing them.

Returns the return code of the executed package manager command.

install_substitutes (**args*, ***kwargs*)
Will try to call the install() method with several lists of packages passed as a variable number of parameters. This is useful if, for example, the names of the packages are different from one distro or distro version to another. For example, `libxcb` for `Apt` is named `libxcb-util-dev` in Ubuntu >= 15.0 and `libxcb-util0-dev` for other versions. You can call to:

```
# will install the first list of packages that succeeds in the␣
↪installation
Apt.install_substitutes(["libxcb-util-dev"], ["libxcb-util0-dev"])
```

Parameters

- **packages_alternatives** – try to install the list of packages passed as a parameter.
- **update** – try to update the package manager database before checking and installing.
- **check** – check if the packages are already installed before installing them.

Returns the return code of the executed package manager command.

update (**args*, ***kwargs*)
Update the system package manager database. Its behaviour is affected by the value of `tools.system.package_manager:mode` *configuration*.

Returns the return code of the executed package manager update command.

conan.tools.system.package_manager.PkgUtil

Will invoke the *pkgutil* command. Enabled by default for **Solaris**.

Reference

class PkgUtil (*conanfile*)

Parameters **conanfile** – The current recipe object. Always use `self`.

check (**args*, ***kwargs*)
Check if the list of packages passed as parameter are already installed.

Parameters **packages** – list of packages to check.

Returns list of packages from the packages argument that are not installed in the system.

install (**args*, ***kwargs*)
Will try to install the list of packages passed as a parameter. Its behaviour is affected by the value of `tools.system.package_manager:mode` *configuration*.

Parameters

- **packages** – try to install the list of packages passed as a parameter.
- **update** – try to update the package manager database before checking and installing.
- **check** – check if the packages are already installed before installing them.

Returns the return code of the executed package manager command.

install_substitutes(**args*, ***kwargs*)

Will try to call the install() method with several lists of packages passed as a variable number of parameters. This is useful if, for example, the names of the packages are different from one distro or distro version to another. For example, `libxcb` for `Apt` is named `libxcb-util-dev` in Ubuntu >= 15.0 and `libxcb-util0-dev` for other versions. You can call to:

```
# will install the first list of packages that succeeds in the␣
↪installation
Apt.install_substitutes(["libxcb-util-dev"], ["libxcb-util0-dev"])
```

Parameters

- **packages_alternatives** – try to install the list of packages passed as a parameter.
- **update** – try to update the package manager database before checking and installing.
- **check** – check if the packages are already installed before installing them.

Returns the return code of the executed package manager command.

update(**args*, ***kwargs*)

Update the system package manager database. Its behaviour is affected by the value of `tools.system.package_manager:mode` *configuration*.

Returns the return code of the executed package manager update command.

conan.tools.system.package_manager.Chocolatey

Will invoke the *choco* command. Enabled by default for **Windows**.

Reference

class Chocolatey(*conanfile*)

Parameters **conanfile** – The current recipe object. Always use `self`.

check(**args*, ***kwargs*)

Check if the list of packages passed as parameter are already installed.

Parameters **packages** – list of packages to check.

Returns list of packages from the packages argument that are not installed in the system.

install(**args*, ***kwargs*)

Will try to install the list of packages passed as a parameter. Its behaviour is affected by the value of `tools.system.package_manager:mode` *configuration*.

Parameters

- **packages** – try to install the list of packages passed as a parameter.
- **update** – try to update the package manager database before checking and installing.
- **check** – check if the packages are already installed before installing them.

Returns the return code of the executed package manager command.

install_substitutes(**args*, ***kwargs*)

Will try to call the install() method with several lists of packages passed as a variable number of parameters. This is useful if, for example, the names of the packages are different from one distro or distro version to another. For example, `libxcb` for `Apt` is named `libxcb-util-dev` in Ubuntu >= 15.0 and `libxcb-util0-dev` for other versions. You can call to:

```
# will install the first list of packages that succeeds in the␣
↪installation
Apt.install_substitutes(["libxcb-util-dev"], ["libxcb-util0-dev"])
```

Parameters

- **packages_alternatives** – try to install the list of packages passed as a parameter.
- **update** – try to update the package manager database before checking and installing.
- **check** – check if the packages are already installed before installing them.

Returns the return code of the executed package manager command.

update(**args*, ***kwargs*)

Update the system package manager database. Its behaviour is affected by the value of `tools.system.package_manager:mode` *configuration*.

Returns the return code of the executed package manager update command.

7.4.9 conan.tools.microsoft

MSBuild

The `MSBuild` build helper is a wrapper around the command line invocation of MSBuild. It abstracts the calls like `msbuild "MyProject.sln" /p:Configuration=<conf> /p:Platform=<platform>` into Python method ones.

This helper can be used like:

```
from conan import ConanFile
from conan.tools.microsoft import MSBuild

class App(ConanFile):
    settings = "os", "arch", "compiler", "build_type"

    def build(self):
        msbuild = MSBuild(self)
        msbuild.build("MyProject.sln")
```

The `MSBuild.build()` method internally implements a call to `msbuild` like:

```
$ <vcvars-cmd> && msbuild "MyProject.sln" /p:Configuration=<configuration> /
↪p:Platform=<platform>
```

Where:

- `<vcvars-cmd>` calls the Visual Studio prompt that matches the current recipe `settings`.
- `configuration`, typically Release, Debug, which will be obtained from `settings.build_type` but this can be customized with the `build_type` attribute.

- `<platform>` is the architecture, a mapping from the `settings.arch` to the common 'x86', 'x64', 'ARM', 'ARM64'. This can be customized with the `platform` attribute.

Customization

attributes

You can customize the following attributes in case you need to change them:

- **build_type** (default `settings.build_type`): Value for the `/p:Configuration`.
- **platform** (default based on `settings.arch` to select one of these values: (`'x86', 'x64', 'ARM', 'ARM64'`): Value for the `/p:Platform`.

Example:

```
from conan import ConanFile
from conan.tools.microsoft import MSBuild
class App(ConanFile):
    settings = "os", "arch", "compiler", "build_type"
    def build(self):
        msbuild = MSBuild(self)
        msbuild.build_type = "MyRelease"
        msbuild.platform = "MyPlatform"
        msbuild.build("MyProject.sln")
```

conf

`MSBuild` is affected by these `[conf]` variables:

- `tools.microsoft.msbuild:verbosity` accepts one of `"Quiet", "Minimal", "Normal", "Detailed", "Diagnostic"` to be passed to the `MSBuild.build()` call as `msbuild .... /verbosity:XXX`.
- `tools.microsoft.msbuild:max_cpu_count` maximum number of CPUs to be passed to the `MSBuild.build()` call as `msbuild .... /m:N`.

Reference

class MSBuild(*conanfile*)

MSBuild build helper class

Parameters conanfile – `< ConanFile object >` The current recipe object. Always use `self`.

command(*sln*, *targets=None*)

Gets the `msbuild` command line. For instance, **`msbuild "MyProject.sln" /p:Configuration=<conf> /p:Platform=<platform>`**.

Parameters

- **sln** – `str` name of Visual Studio `*.sln` file
- **targets** – `targets` is an optional argument, defaults to `None`, and otherwise it is a list of targets to build

Returns `str` msbuild command line.

build(*sln*, *targets=None*)
Runs the `msbuild` command line obtained from `self.command(sln)`.

Parameters

- **sln** – `str` name of Visual Studio `*.sln` file
- **targets** – `targets` is an optional argument, defaults to `None`, and otherwise it is a list of targets to build

MSBuildDeps

The `MSBuildDeps` is the dependency information generator for Microsoft MSBuild build system. It will generate multiple *xxxx.props* properties files, one per dependency of a package, to be used by consumers using MSBuild or Visual Studio, just adding the generated properties files to the solution and projects.

The `MSBuildDeps` generator can be used by name in conanfiles:

Listing 40: conanfile.py

```
class Pkg(ConanFile):
    generators = "MSBuildDeps"
```

Listing 41: conanfile.txt

```
[generators]
MSBuildDeps
```

And it can also be fully instantiated in the conanfile `generate()` method:

Listing 42: conanfile.py

```
from conan import ConanFile
from conan.tools.microsoft import MSBuildDeps

class Pkg(ConanFile):
    settings = "os", "compiler", "arch", "build_type"
    requires = "zlib/1.2.11", "bzip2/1.0.8"

    def generate(self):
        ms = MSBuildDeps(self)
        ms.generate()
```

Generated files

The `MSBuildDeps` generator is a multi-configuration generator, and generates different files for any different Debug/Release configuration. For instance, running these commands:

```
$ conan install .  # default is Release
$ conan install . -s build_type=Debug
```

It generates the next files:

- *conan_zlib_vars_release_x64.props*: `Conanzlibxxxx` variables definitions for the `zlib` dependency, Release config, like `ConanzlibIncludeDirs`, `ConanzlibLibs`, etc.

- *conan_zlib_vars_debug_x64.props*: Same `Conanzlib``variables for ``zlib` dependency, Debug config
- *conan_zlib_release_x64.props*: Activation of `Conanzlibxxxx` variables in the current build as standard C/C++ build configuration, Release config. This file contains also the transitive dependencies definitions.
- *conan_zlib_debug_x64.props*: Same activation of `Conanzlibxxxx` variables, Debug config, also inclusion of transitive dependencies.
- *conan_zlib.props*: Properties file for `zlib`. It conditionally includes, depending on the configuration, one of the two immediately above Release/Debug properties files.
- Same 5 files are generated for every dependency in the graph, in this case `conan_bzip.props` too, which conditionally includes the Release/Debug bzip properties files.
- *conandeps.props*: Properties files that includes all direct dependencies, for this case `conan_zlib.props` and `conan_bzip2.props`

Add the *conandeps.props* to your solution project files if you want to depend on all the declared dependencies. For single project solutions, this is probably the way to go. For multi-project solutions, you might be more efficient and add properties files per project. You could add *conan_zlib.props* properties to "project1" in the solution and *conan_bzip2.props* to "project2" in the solution for example.

The above files are generated when the package doesn't have components. If the package has defined components, the following files will be generated:

- *conan_pkgname_compname_vars_release_x64.props*: Definition of variables for the component `compname` of the package `pkgname`
- *conan_pkgname_compname_release_x64.props*: Activation of the above variables into VS effective variables to be used in the build
- *conan_pkgname_compname.props*: Properties file for component `compname` of package `pkgname`. It conditionally includes, depending on the configuration, the specific activation property files.
- *conan_pkgname.props*: Properties file for package `pkgname`. It includes and aggregates all the components of the package.
- *conandeps.props*: Same as above, aggregates all the direct dependencies property files for the packages (like `conan_pkgname.props`)

If your project depends only on certain components, the specific `conan_pkgname_compname.props` files can be added to the project instead of the global or the package ones.

Requirement traits support

The above generated files, more specifically the files containing the variables (`conan_pkgname_vars_release_x64.props/conan_pkgname_compname_vars_release_x64.props`), will not contain all the information if the requirement traits have excluded them. For example, by default, the `includedirs` of transitive dependencies will be empty, as those headers shouldn't be included by the user unless a specific `requires` to that package is defined.

Configurations

If your Visual Studio project defines custom configurations, like `ReleaseShared`, or `MyCustomConfig`, it is possible to define it into the `MSBuildDeps` generator, so different project configurations can use different set of dependencies. Let's say that our current project can be built as a shared library, with the custom configuration `ReleaseShared`, and the package also controls this with the `shared` option:

```
from conan import ConanFile
from conan.tools.microsoft import MSBuildDeps

class Pkg(ConanFile):
    settings = "os", "compiler", "arch", "build_type"
    options = {"shared": [True, False]}
    default_options = {"shared": False}
    requires = "zlib/1.2.11"

    def generate(self):
        ms = MSBuildDeps(self)
        # We assume that -o *:shared=True is used to install all shared deps too
        if self.options.shared:
            ms.configuration = str(self.settings.build_type) + "Shared"
        ms.generate()
```

This generates new properties files for this custom configuration, and switching it in the IDE allows to gather dependencies configuration like Debug/Release, and even static and/or shared libraries.

Dependencies

`MSBuildDeps` uses the `self.dependencies` to access to the dependencies information. The following dependencies are translated to properties files:

- All the direct dependencies, which are the ones declared by the current `conanfile`, live in the `host` context: all regular `requires`, plus the `tool_requires`, that are in the host context, e.g. test frameworks like `gtest` or `catch`.
- All transitive `requires` of those direct dependencies (all in the host context)
- Tool requires, in the build context, that is, application and executables that run in the build machine irrespective of the destination platform, are added exclusively to the `<ExecutablePath>` property, taking the value from `$(Conan{{name}}BinaryDirectories)` defined properties. This allows to define custom build commands, invoke code generation tools, with the `<CustomBuild>` and `<Command>` elements.

Customization

conf

`MSBuildDeps` is affected by these `[conf]` variables:

- `tools.microsoft.msbuilddeps:exclude_code_analysis` list of packages names patterns to be added to the Visual Studio `CAExcludePath` property.

Reference

class MSBuildDeps(*conanfile*)

MSBuildDeps class generator conandeps.props: unconditional import of all *direct* dependencies only

Parameters **conanfile** – `< ConanFile object >` The current recipe object. Always use `self`.

generate()
Generates `conan_<pkg>_<config>_vars.props`, `conan_<pkg>_<config>.props`, and `conan_<pkg>.props` files into the `conanfile.generators_folder`.

MSBuildToolchain

The `MSBuildToolchain` is the toolchain generator for MSBuild. It will generate MSBuild properties files that can be added to the Visual Studio solution projects. This generator translates the current package configuration, settings, and options, into MSBuild properties files syntax.

This generator can be used by name in conanfiles:

Listing 43: **conanfile.py**

```
class Pkg(ConanFile):
    generators = "MSBuildToolchain"
```

Listing 44: **conanfile.txt**

```
[generators]
MSBuildToolchain
```

And it can also be fully instantiated in the conanfile `generate()` method:

```
from conan import ConanFile
from conan.tools.microsoft import MSBuildToolchain

class App(ConanFile):
    settings = "os", "arch", "compiler", "build_type"

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

The `MSBuildToolchain` will generate three files after a `conan install` command:

```
$ conan install . # default is Release
$ conan install . -s build_type=Debug
```

- The main *conantoolchain.props* file, to be added to the project.
- A *conantoolchain_<config>.props* file, that will be conditionally included from the previous *conantoolchain.props* file based on the configuration and platform, e.g., *conantoolchain_release_x86.props*.
- A *conanvcvars.bat* file with the `vcvars` invocation to define the build environment from the command line, or any other automated tools (might not be required if opening the IDE). This file will be automatically called by the `MSBuild.build()` method.

Every invocation with different configuration creates a new properties `.props` file, that is also conditionally included. That allows to install different configurations, then switch among them directly from the Visual Studio IDE.

The MSBuildToolchain files can configure:

- The Visual Studio runtime (*MT/MD/MTd/MDd*), obtained from Conan input settings.
- The C++ standard, obtained from Conan input settings.

One of the advantages of using toolchains is that they help to achieve the exact same build with local development flows, than when the package is created in the cache.

Customization

conf

`MSBuildToolchain` is affected by these `[conf]` variables:

- `tools.microsoft.msbuildtoolchain:compile_options` dict-like object of extra compile options to be added to `<ClCompile>` section. The dict will be translated as follows: `<[KEY]>[VALUE]</[KEY]>`.
- `tools.build:cxxflags` list of extra C++ flags that will be appended to `<AdditionalOptions>` section from `<ClCompile>` and `<ResourceCompile>` one.
- `tools.build:cflags` list of extra of pure C flags that will be appended to `<AdditionalOptions>` section from `<ClCompile>` and `<ResourceCompile>` one.
- `tools.build:sharedlinkflags` list of extra linker flags that will be appended to `<AdditionalOptions>` section from `<Link>` one.
- `tools.build:exelinkflags` list of extra linker flags that will be appended to `<AdditionalOptions>` section from `<Link>` one.
- `tools.build:defines` list of preprocessor definitions that will be appended to `<PreprocessorDefinitions>` section from `<ResourceCompile>` one.

Reference

class MSBuildToolchain(*conanfile*)

MSBuildToolchain class generator

Parameters conanfile – `< ConanFile object >` The current recipe object. Always use `self`.

generate()

Generates a `conantoolchain.props`, a `conantoolchain_<config>.props`, and, if `compiler=msvc`, a `conanvcvars.bat` files. In the first two cases, they'll have the valid XML format with all the good settings like any other VS project `*.props` file. The last one emulates the `vcvarsall.bat` env script. See also *VCVars*.

Attributes

- **properties**: Additional properties added to the generated `.props` files. You can define the properties in a key-value syntax like:

```python
from conan import ConanFile
from conan.tools.microsoft import MSBuildToolchain

class App(ConanFile):
    settings = "os", "arch", "compiler", "build_type"

    def generate(self):
        msbuild = MSBuildToolchain(self)
        msbuild.properties["IncludeExternals"] = "true"
        msbuild.generate()
```

Then, the generated *conantoolchain_<config>.props* file will contain the defined property in its contents:

```xml
<?xml version="1.0" encoding="utf-8"?>
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
<ItemDefinitionGroup>
...
</ItemDefinitionGroup>
<PropertyGroup Label="Configuration">
    ...
    <IncludeExternals>true</IncludeExternals>
    ...
</PropertyGroup>
</Project>
```

VCVars

Generates a file called `conanvcvars.bat` that activates the Visual Studio developer command prompt according to the current settings by wrapping the vcvarsall Microsoft bash script.

The `VCVars` generator can be used by name in conanfiles:

Listing 45: conanfile.py

```python
class Pkg(ConanFile):
    generators = "VCVars"
```

Listing 46: conanfile.txt

```
[generators]
VCVars
```

And it can also be fully instantiated in the conanfile `generate()` method:

Listing 47: conanfile.py

```python
from conan import ConanFile
from conan.tools.microsoft import VCVars

class Pkg(ConanFile):
    settings = "os", "compiler", "arch", "build_type"
    requires = "zlib/1.2.11", "bzip2/1.0.8"

    def generate(self):
        ms = VCVars(self)
        ms.generate()
```

Customization

conf

`VCVars` is affected by these `[conf]` variables:

- `tools.microsoft.msbuild:installation_path` indicates the path to Visual Studio installation folder. For instance: `C:\Program Files (x86)\Microsoft Visual Studio\2019\Community`, `C:\Program Files (x86)\Microsoft Visual Studio 14.0`, etc.

Reference

class VCVars(*conanfile*)

VCVars class generator

Parameters conanfile – < `ConanFile object` > The current recipe object. Always use `self`.

generate(*scope='build'*)

Creates a `conanvcvars.bat` file with the good args from settings to set environment variables to configure the command line for native 32-bit or 64-bit compilation.

Parameters scope – `str` Launcher to be used to run all the variables. For instance, if `build`, then it'll be used the `conanbuild` launcher.

NMakeDeps

This generator can be used as:

```
from conan import ConanFile

class Pkg(ConanFile):
    settings = "os", "compiler", "build_type", "arch"

    requires = "mydep/1.0"
    # attribute declaration
    generators = "NMakeDeps"

    # OR explicit usage in the generate() method
    def generate(self):
        deps = NMakeDeps(self)
        deps.generate()

    def build(self):
        self.run(f"nmake /f makefile")
```

The generator will create a `conannmakedeps.bat` environment script that defines `CL`, `LIB` and `_LINK_` environment variables, injecting necessary flags to locate and link the dependencies declared in `requires`. This generator should most likely be used together with `NMakeToolchain` one.

NMakeToolchain

This generator can be used as:

```
from conan import ConanFile

class Pkg(ConanFile):
    settings = "os", "compiler", "build_type", "arch"
    generators = "NMakeToolchain"

    def build(self):
        self.run("nmake /f makefile")
```

Or it can be fully instantiated in the conanfile `generate()` method:

```
from conan import ConanFile
from conan.tools.microsoft import NMakeToolchain

class Pkg(ConanFile):
    settings = "os", "arch", "compiler", "build_type"

    def generate(self):
        tc = NMakeToolchain(self)
        tc.generate()

    def build(self):
        self.run("nmake /f makefile")
```

NMakeToolchain generator will create a `conannmaketoolchain.bat` environment script injecting flags deduced from profile (build_type, runtime, cppstd, build flags from conf) into environment variables NMake can understand: `CL` and `_LINK_`. It will also generate a `conanvcvars.bat` script that activates the correct VS prompt matching the Conan host settings `arch`, `compiler` and `compiler.version`, and build settings `arch`.

constructor

```
def __init__(self, conanfile):
```

- `conanfile`: the current recipe object. Always use `self`.

Attributes

You can change some attributes before calling the `generate()` method if you want to inject more flags:

```
from conan import ConanFile
from conan.tools.microsoft import NMakeToolchain

class Pkg(ConanFile):
    settings = "os", "arch", "compiler", "build_type"

    def generate(self):
        tc = NMakeToolchain(self)
        tc.extra_cflags.append("/my_flag")
        tc.extra_defines.append("FOO=BAR")
        tc.generate()
```

- **extra_cflags** (Defaulted to `[]`): Additional cflags.
- **extra_cxxflags** (Defaulted to `[]`): Additional cxxflags.
- **extra_defines** (Defaulted to `[]`): Additional defines.
- **extra_ldflags** (Defaulted to `[]`): Additional ldflags.

conf

`NMaketoolchain` is affected by these `[conf]` variables:

- `tools.build:cflags` list of extra pure C flags that will be used by `CL`.
- `tools.build:cxxflags` list of extra C++ flags that will be used by `CL`.

- `tools.build:defines` list of preprocessor definitions that will be used by `CL`.
- `tools.build:sharedlinkflags` list of extra linker flags that will be used by `_LINK_`.
- `tools.build:exelinkflags` list of extra linker flags that will be used by `_LINK_`.
- `tools.build:compiler_executables` dict-like Python object which specifies the compiler as key and the compiler executable path as value. Those keys will be mapped as follows:
  - `asm`: will set `AS` in *conannmaketoolchain.sh|bat* script.
  - `c`: will set `CC` in *conannmaketoolchain.sh|bat* script.
  - `cpp`: will set `CPP` and `CXX` in *conannmaketoolchain.sh|bat* script.
  - `rc`: will set `RC` in *conannmaketoolchain.sh|bat* script.

Customizing the environment

If your `Makefile` script needs some other environment variable rather than `CL` and `_LINK_`, you can customize it before calling the `generate()` method. Call the `environment()` method to calculate the mentioned variables and then add the variables that you need. The `environment()` method returns an *Environment* object:

```
from conan import ConanFile
from conan.tools.microsoft import NMakeToolchain

class Pkg(ConanFile):
    settings = "os", "arch", "compiler", "build_type"

    def generate(self):
        tc = NMakeToolchain(self)
        env = tc.environment()
        env.define("FOO", "BAR")
        tc.generate(env)
```

You can also inspect default environment variables NMakeToolchain will inject in *conannmaketoolchain.sh|bat* script:

```
from conan import ConanFile
from conan.tools.microsoft import NMakeToolchain

class Pkg(ConanFile):
    settings = "os", "arch", "compiler", "build_type"

    def generate(self):
        tc = NMakeToolchain(self)
        env_vars = tc.vars()
        cl_env_var = env_vars.get("CL")
```

vs_layout

vs_layout(*conanfile*)

Initialize a layout for a typical Visual Studio project.

Parameters **conanfile** – < `ConanFile object` > The current recipe object. Always use `self`.

conan.tools.microsoft.visual

check_min_vs

check_min_vs(*conanfile*, *version*, *raise_invalid=True*)

This is a helper method to allow the migration of 1.X -> 2.0 and VisualStudio -> msvc settings without breaking recipes. The legacy "Visual Studio" with different toolset is not managed, not worth the complexity.

Parameters

- **raise_invalid** – `bool` Whether to raise or return False if the version check fails
- **conanfile** – `< ConanFile object >` The current recipe object. Always use `self`.
- **version** – `str` Visual Studio or msvc version number.

Example:

```
def validate(self):
    check_min_vs(self, "192")
```

msvc_runtime_flag

msvc_runtime_flag(*conanfile*)

Gets the MSVC runtime flag given the `compiler.runtime` value from the settings.

Parameters **conanfile** – `< ConanFile object >` The current recipe object. Always use `self`.

Returns `str` runtime flag.

is_msvc

is_msvc(*conanfile*, *build_context=False*)

Validates if the current compiler is `msvc`.

Parameters

- **conanfile** – `< ConanFile object >` The current recipe object. Always use `self`.
- **build_context** – If True, will use the settings from the build context, not host ones

Returns `bool` True, if the host compiler is `msvc`, otherwise, False.

is_msvc_static_runtime

is_msvc_static_runtime(*conanfile*)

Validates when building with Visual Studio or msvc and MT on runtime.

Parameters **conanfile** – `< ConanFile object >` The current recipe object. Always use `self`.

Returns `bool` True, if `msvc + runtime MT`. Otherwise, False.

msvs_toolset

msvs_toolset(*conanfile*)
Returns the corresponding platform toolset based on the compiler of the given conanfile. In case no toolset is configured in the profile, it will return a toolset based on the compiler version, otherwise, it will return the toolset from the profile. When there is no compiler version neither toolset configured, it will return None It supports Visual Studio, msvc and Intel.

Parameters **conanfile** – Conanfile instance to access settings.compiler

Returns A toolset when compiler.version is valid or compiler.toolset is configured. Otherwise, None.

conan.tools.microsoft.subsystems

unix_path

unix_path(*conanfile*, *path*, *scope='build'*)

7.4.10 conan.tools.scm

Git

class Git(*conanfile*, *folder='.'*)
Git is a wrapper for several common patterns used with *git* tool.

Parameters

- **conanfile** – Conanfile instance.
- **folder** – Current directory, by default **.**, the current working directory.

run(*cmd*)
Executes `git <cmd>`

Returns The console output of the command.

get_commit()

Returns The current commit, with `git rev-list HEAD -n 1 -- <folder>`. The latest commit is returned, irrespective of local not committed changes.

get_remote_url(*remote='origin'*)
Obtains the URL of the remote git remote repository, with `git remote -v`

Warning! Be aware that This method will get the output from `git remote -v`. If you added tokens or credentials to the remote in the URL, they will be exposed. Credentials shouldn't be added to git remotes definitions, but using a credentials manager or similar mechanism. If you still want to use this approach, it is your responsibility to strip the credentials from the result.

Parameters **remote** – Name of the remote git repository ('origin' by default).

Returns URL of the remote git remote repository.

commit_in_remote(*commit*, *remote='origin'*)
Checks that the given commit exists in the remote, with `branch -r --contains <commit>` and checking an occurrence of a branch in that remote exists.

Parameters

- **commit** – Commit to check.
- **remote** – Name of the remote git repository ('origin' by default).

Returns True if the given commit exists in the remote, False otherwise.

is_dirty()

Returns if the current folder is dirty, running `git status -s`

Returns True, if the current folder is dirty. Otherwise, False.

get_url_and_commit(*remote='origin'*)

This is an advanced method, that returns both the current commit, and the remote repository url. This method is intended to capture the current remote coordinates for a package creation, so that can be used later to build again from sources from the same commit. This is the behavior:

- If the repository is dirty, it will raise an exception. Doesn't make sense to capture coordinates of something dirty, as it will not be reproducible. If there are local changes, and the user wants to test a local conan create, should commit the changes first (locally, not push the changes).
- If the repository is not dirty, but the commit doesn't exist in the given remote, the method will return that commit and the URL of the local user checkout. This way, a package can be conan create created locally, testing everything works, before pushing some changes to the remote.
- If the repository is not dirty, and the commit exists in the specified remote, it will return that commit and the url of the remote.

Warning! Be aware that This method will get the output from `git remote -v`. If you added tokens or credentials to the remote in the URL, they will be exposed. Credentials shouldn't be added to git remotes definitions, but using a credentials manager or similar mechanism. If you still want to use this approach, it is your responsibility to strip the credentials from the result.

Parameters **remote** – Name of the remote git repository ('origin' by default).

Returns (url, commit) tuple

get_repo_root()

Get the current repository top folder with `git rev-parse --show-toplevel`

Returns Repository top folder.

clone(*url*, *target=''*, *args=None*)

Performs a `git clone <url> <args> <target>` operation, where target is the target directory.

Parameters

- **url** – URL of remote repository.
- **target** – Target folder.
- **args** – Extra arguments to pass to the git clone as a list.

fetch_commit(*url*, *commit*)

Experimental: does a 1 commit fetch and checkout, instead of a full clone, should be faster.

checkout(*commit*)

Checkouts the given commit using `git checkout <commit>`.

Parameters **commit** – Commit to checkout.

included_files()

Run `git ls-files --full-name --others --cached --exclude-standard` to the get the list
of files not ignored by `.gitignore`

Returns List of files.

Version

class Version(*value*)

This is NOT an implementation of semver, as users may use any pattern in their versions. It is just a helper to parse "." or "-" and compare taking into account integers when possible

bump(*index*)

Meta private Bump the version Increments by 1 the version field at the specified index, setting to 0 the fields on the right. 2.5 => bump(1) => 2.6 1.5.7 => bump(0) => 2.0.0

Parameters **index** –

7.4.11 conan.tools.layout

Predefined layouts

There are some pre-defined common *layouts*, ready to be simply used in recipes:

- `cmake_layout()`: *a layout for a typical CMake project*
- `vs_layout()`: a layout for a typical Visual Studio project
- `basic_layout()`: *a very basic layout for a generic project*

The pre-defined layouts define the Conanfile `.folders` and `.cpp` attributes with typical values. To check which values are set by these pre-defined layouts please check the reference for the *layout()* method. For example in the `cmake_layout()` the source folder is set to `"."`, meaning that Conan will expect the sources in the same directory where the conanfile is (most likely the project root, where a `CMakeLists.txt` file will be typically found). If you have a different folder where the `CMakeLists.txt` is located, you can use the `src_folder` argument:

```
from conan.tools.cmake import cmake_layout

def layout(self):
    cmake_layout(self, src_folder="mysrcfolder")
```

Even if this pre-defined layout doesn't suit your specific projects layout, checking how they implement their logic shows how you could implement your own logic (and probably put it in a common `python_require` if you are going to use it in multiple packages).

To learn more about the layouts and how to use them while developing packages, please check the Conan package layout *tutorial*.

basic_layout

Usage:

```
from conan.tools.layout import basic_layout

def layout(self):
    basic_layout(self)
```

The current layout implementation is very simple, basically sets a different build folder for different build_types and sets the generators output folder inside the build folder. This way we avoid to clutter our project while working locally.

```
def basic_layout(conanfile, src_folder="."):
    conanfile.folders.build = "build"
    if conanfile.settings.get_safe("build_type"):
        conanfile.folders.build += "-{}".format(str(conanfile.settings.build_type).
↪lower())
    conanfile.folders.generators = os.path.join(conanfile.folders.build, "conan")
    conanfile.cpp.build.bindirs = ["."]
    conanfile.cpp.build.libdirs = ["."]
    conanfile.folders.source = src_folder
```

7.4.12 conan.tools.intel

IntelCC

This tool helps you to manage the new Intel oneAPI DPC++/C++ and Classic ecosystem in Conan.

Warning: This generator is **experimental** and subject to breaking changes.

Warning: macOS is not supported for the Intel oneAPI DPC++/C++ (icx/icpx or dpcpp) compilers. For macOS or Xcode support, you'll have to use the Intel C++ Classic Compiler.

Note: Remember, you need to have installed previously the Intel oneAPI software.

This generator creates a `conanintelsetvars.sh|bat` wrapping the Intel script `setvars.sh|bat` that sets the Intel oneAPI environment variables needed. That script is the first step to start using the Intel compilers because it's setting some important variables in your local environment.

In summary, the `IntelCC` generator:

1. Reads your profile `[settings]` and `[conf]`.
2. Uses that information to generate a `conanintelsetvars.sh|bat` script with the command to load the Intel `setvars.sh|bat` script.
3. Then, you or the chosen generator will be able to run that script and use any Intel compiler to compile the project.

Note: You can launch the `conanintelsetvars.sh|bat` before calling your intel compiler to build a project. Conan will also call it in the conanfile `build(self)` method when running any command with `self.run`.

At first, ensure you are using a *profile* like this one:

Listing 48: *intelprofile*

```
[settings]
...
compiler=intel-cc
compiler.mode=dpcpp
compiler.version=2021.3
```

(continues on next page)

(continued from previous page)

```
compiler.libcxx=libstdc++
build_type=Release

[buildenv]
CC=dpcpp
CXX=dpcpp

[conf]
tools.intel:installation_path=/opt/intel/oneapi
```

The `IntelCC` generator can be used by name in conanfiles:

Listing 49: *conanfile.py*

```python
class Pkg(ConanFile):
    generators = "IntelCC"
```

Listing 50: *conanfile.txt*

```
[generators]
IntelCC
```

And it can also be fully instantiated in the conanfile `generate()` method:

Listing 51: *conanfile.py*

```python
from conan import ConanFile
from conan.tools.intel import IntelCC

class App(ConanFile):
    settings = "os", "arch", "compiler", "build_type"

    def generate(self):
        intelcc = IntelCC(self)
        intelcc.generate()
```

Now, running the command **`conan install . -pr intelprofile`** generates the `conanintelsetvars.sh|bat` script which runs the Intel *setvars* script and loads all the variables into your local environment.

Custom configurations

Apply different installation paths and command arguments simply by changing the `[conf]` entries. For instance:

Listing 52: intelprofile

```
[settings]
...
compiler=intel-cc
compiler.mode=dpcpp
compiler.version=2021.3
compiler.libcxx=libstdc++
build_type=Release

[buildenv]
CC=dpcpp
```

(continues on next page)

(continued from previous page)

```
CXX=dpcpp

[conf]
tools.intel:installation_path=/opt/intel/oneapi
tools.intel:setvars_args=--config="full/path/to/your/config.txt" --force
```

Run again a **conan install . -pr intelprofile**, then the `conanintelsetvars.sh` script (if we are using Linux OS) will contain something like:

Listing 53: conanintelsetvars.sh

```
. "/opt/intel/oneapi/setvars.sh" --config="full/path/to/your/config.txt" --force
```

Reference

class IntelCC(*conanfile*)

Class that manages Intel oneAPI DPC++/C++/Classic Compilers vars generation

arch = None

arch setting

ms_toolset

Get Microsoft Visual Studio Toolset depending on the mode selected

generate(*scope='build'*)

Generate the Conan Intel file to be loaded in build environment by default

installation_path

Get the Intel oneAPI installation root path

command

The Intel oneAPI DPC++/C++ Compiler includes environment configuration scripts to configure your build and development environment variables:

- On Linux, the file is a shell script called setvars.sh.
- On Windows, the file is a batch file called setvars.bat.
- Linux -> `>> . /<install-dir>/setvars.sh <arg1> <arg2> ... <argn><arg1> <arg2> ... <argn>` The compiler environment script file accepts an optional target architecture argument <arg>: - intel64: Generate code and use libraries for Intel 64 architecture-based targets. - ia32: Generate code and use libraries for IA-32 architecture-based targets.
- Windows -> `>> call <install-dir>\setvars.bat [<arg1>] [<arg2>]` Where <arg1> is optional and can be one of the following: - intel64: Generate code and use libraries for Intel 64 architecture (host and target). - ia32: Generate code and use libraries for IA-32 architecture (host and target).

  With the dpcpp compiler, <arg1> is intel64 by default.

  The <arg2> is optional. If specified, it is one of the following: - vs2019: Microsoft Visual Studio* 2019 - vs2017: Microsoft Visual Studio 2017

  Returns *str* setvars.sh|bat command to be run

conf

`IntelCC` uses these *configuration entries*:

- `tools.intel:installation_path`: **(required)** argument to tell Conan the installation path, if it's not defined, Conan will try to find it out automatically.
- `tools.intel:setvars_args`: **(optional)** it is used to pass whatever we want as arguments to our *setvars.sh|bat* file. You can check out all the possible ones from the Intel official documentation.

7.4.13 conan.tools.android

android_abi()

android_abi(*conanfile*, *context='host'*)
Returns Android-NDK ABI

Parameters

- **conanfile** – ConanFile instance
- **context** – either "host", "build" or "target"

Returns Android-NDK ABI

This function might not be necessary when using Conan built-in integrations, as they already manage it, but can be useful if developing your own build system integration.

`android_abi()` function returns the Android standard ABI name based on Conan `settings.arch` value, something like:

```
def android_abi(conanfile, context="host"):
  ...
  return {
        "armv5el": "armeabi",
        "armv5hf": "armeabi",
        "armv5": "armeabi",
        "armv6": "armeabi-v6",
        "armv7": "armeabi-v7a",
        "armv7hf": "armeabi-v7a",
        "armv8": "arm64-v8a",
        }.get(conanfile.settings.arch)
```

As it can be seen, the default is the "host" ABI, but it is possible to select also the "build" or "target" ones if necessary.

```
from conan.tools.android import android_abi

class Pkg(ConanFile):
    def generate(self)
        abi = android_abi(self)
```

7.5 Configuration files

These are the most important configuration files, used to customize conan.

7.5.1 global.conf

The **global.conf** file is located in the Conan user home directory, e.g., *[CONAN_HOME]/global.conf*.

Introduction to configuration

global.conf is aimed to save some core/tools/user configuration variables that will be used by Conan. For instance:

- Package ID modes.
- General HTTP(python-requests) configuration.
- Number of retries when downloading/uploading recipes.
- Related tools configurations (used by toolchains, helpers, etc.)
- Others (required Conan version, CLI non-interactive, etc.)

Let's briefly explain the three types of existing configurations:

- `core.*`: aimed to configure values of Conan core behavior (download retries, package ID modes, etc.). Only definable in *global.conf* file.
- `tools.*`: aimed to configure values of Conan tools (toolchains, build helpers, etc.) used in your recipes. Definable in both *global.conf* and *profiles*.
- `user.*`: aimed to define personal user configurations. They can define whatever user wants. Definable in both *global.conf* and *profiles*.

To list all the possible configurations available, run **conan config list**:

```
$ conan config list

core.cache:storage_path: Absolute path where the packages and database are stored
core.download:download_cache: Define path to a file download cache
core.download:parallel: Number of concurrent threads to download packages
core.download:retry: Number of retries in case of failure when downloading from Conan␣
↪server
core.download:retry_wait: Seconds to wait between download attempts from Conan server
core.gzip:compresslevel: The Gzip compresion level for Conan artifacts (default=9)
core.net.http:cacert_path: Path containing a custom Cacert file
core.net.http:clean_system_proxy: If defined, the proxies system env-vars will be␣
↪discarded
core.net.http:client_cert: Path or tuple of files containing a client cert (and key)
core.net.http:max_retries: Maximum number of connection retries (requests library)
core.net.http:no_proxy_match: List of urls to skip from proxies configuration
core.net.http:proxies: Dictionary containing the proxy configuration
core.net.http:timeout: Number of seconds without response to timeout (requests␣
↪library)
core.package_id:default_build_mode: By default, 'None'
core.package_id:default_embed_mode: By default, 'full_mode'
core.package_id:default_non_embed_mode: By default, 'minor_mode'
core.package_id:default_python_mode: By default, 'minor_mode'
core.package_id:default_unknown_mode: By default, 'semver_mode'
core.upload:retry: Number of retries in case of failure when uploading to Conan server
core.upload:retry_wait: Seconds to wait between upload attempts to Conan server
core:allow_uppercase_pkg_names: Temporarily (will be removed in 2.X) allow uppercase␣
↪names
core:default_build_profile: Defines the default build profile (None by default)
core:default_profile: Defines the default host profile ('default' by default)
```

(continues on next page)

(continued from previous page)

```
core:non_interactive: Disable interactive user input, raises error if input necessary
core:required_conan_version: Raise if current version does not match the defined␣
↪range.
tools.android:ndk_path: Argument for the CMAKE_ANDROID_NDK
tools.apple.xcodebuild:verbosity: Verbosity level for xcodebuild: 'verbose' or 'quiet
tools.apple:enable_arc: (boolean) Enable/Disable ARC Apple Clang flags
tools.apple:enable_bitcode: (boolean) Enable/Disable Bitcode Apple Clang flags
tools.apple:enable_visibility: (boolean) Enable/Disable Visibility Apple Clang flags
tools.apple:sdk_path: Path to the SDK to be used
tools.build.cross_building:can_run: Bool value that indicates whether is possible to␣
↪run a non-native app on the same architecture. It's used by 'can_run' tool
tools.build:cflags: List of extra C flags used by different toolchains like␣
↪CMakeToolchain, AutotoolsToolchain and MesonToolchain
tools.build:compiler_executables: Defines a Python dict-like with the compilers path␣
↪to be used. Allowed keys {'c', 'cpp', 'cuda', 'objc', 'objcxx', 'rc', 'fortran',
↪'asm', 'hip', 'ispc'}
tools.build:cxxflags: List of extra CXX flags used by different toolchains like␣
↪CMakeToolchain, AutotoolsToolchain and MesonToolchain
tools.build:defines: List of extra definition flags used by different toolchains like␣
↪CMakeToolchain and AutotoolsToolchain
tools.build:download_source: Force download of sources for every package
tools.build:exelinkflags: List of extra flags used by CMakeToolchain for CMAKE_EXE_
↪LINKER_FLAGS_INIT variable
tools.build:jobs: Default compile jobs number -jX Ninja, Make, /MP VS (default: max␣
↪CPUs)
tools.build:linker_scripts: List of linker script files to pass to the linker used by␣
↪different toolchains like CMakeToolchain, AutotoolsToolchain, and MesonToolchain
tools.build:sharedlinkflags: List of extra flags used by CMakeToolchain for CMAKE_
↪SHARED_LINKER_FLAGS_INIT variable
tools.build:skip_test: Do not execute CMake.test() and Meson.test() when enabled
tools.build:sysroot: Pass the --sysroot=<tools.build:sysroot> flag if available.␣
↪(None by default)
tools.cmake.cmake_layout:build_folder_vars: Settings and Options that will produce a␣
↪different build folder and different CMake presets names
tools.cmake.cmaketoolchain:find_package_prefer_config: Argument for the CMAKE_FIND_
↪PACKAGE_PREFER_CONFIG
tools.cmake.cmaketoolchain:generator: User defined CMake generator to use instead of␣
↪default
tools.cmake.cmaketoolchain:system_name: Define CMAKE_SYSTEM_NAME in CMakeToolchain
tools.cmake.cmaketoolchain:system_processor: Define CMAKE_SYSTEM_PROCESSOR in␣
↪CMakeToolchain
tools.cmake.cmaketoolchain:system_version: Define CMAKE_SYSTEM_VERSION in␣
↪CMakeToolchain
tools.cmake.cmaketoolchain:toolchain_file: Use other existing file rather than conan_
↪toolchain.cmake one
tools.cmake.cmaketoolchain:toolset_arch: Toolset architecture to be used as part of␣
↪CMAKE_GENERATOR_TOOLSET in CMakeToolchain
tools.cmake.cmaketoolchain:user_toolchain: Inject existing user toolchains at the␣
↪beginning of conan_toolchain.cmake
tools.env.virtualenv:powershell: If it is set to True it will generate powershell␣
↪launchers if os=Windows
tools.files.download:download_cache: Define the cache folder to store downloads from␣
↪files.download()/get()
tools.files.download:retry: Number of retries in case of failure when downloading
tools.files.download:retry_wait: Seconds to wait between download attempts
tools.gnu:define_libcxx11_abi: Force definition of GLIBCXX_USE_CXX11_ABI=1 for␣
↪libstdc++11
```

(continues on next page)

(continued from previous page)

```
tools.gnu:host_triplet: Custom host triplet to pass to Autotools scripts
tools.gnu:make_program: Indicate path to make program
tools.gnu:pkg_config: Path to pkg-config executable used by PkgConfig build helper
tools.google.bazel:bazelrc_path: Defines Bazel rc-path
tools.google.bazel:configs: Define Bazel config file
tools.info.package_id:confs: List of existing configuration to be part of the package␣
↪ID
tools.intel:installation_path: Defines the Intel oneAPI installation root path
tools.intel:setvars_args: Custom arguments to be passed onto the setvars.sh|bat␣
↪script from Intel oneAPI
tools.meson.mesontoolchain:backend: Any Meson backend: ninja, vs, vs2010, vs2012,␣
↪vs2013, vs2015, vs2017, vs2019, xcode
tools.meson.mesontoolchain:extra_machine_files: List of paths for any additional␣
↪native/cross file references to be appended to the existing Conan ones
tools.microsoft.bash:active: If Conan is already running inside bash terminal in␣
↪Windows
tools.microsoft.bash:path: The path to the shell to run when conanfile.win_bash==True
tools.microsoft.bash:subsystem: The subsystem to be used when conanfile.win_
↪bash==True. Possible values: msys2, msys, cygwin, wsl, sfu
tools.microsoft.msbuild:installation_path: VS install path, to avoid auto-detect via␣
↪vswhere, like C:/Program Files (x86)/Microsoft Visual Studio/2019/Community. Use␣
↪empty string to disable
tools.microsoft.msbuild:max_cpu_count: Argument for the /m when running msvc to build␣
↪parallel projects
tools.microsoft.msbuild:verbosity: Verbosity level for MSBuild: 'Quiet', 'Minimal',
↪'Normal', 'Detailed', 'Diagnostic'
tools.microsoft.msbuild:vs_version: Defines the IDE version when using the new msvc␣
↪compiler
tools.microsoft.msbuilddeps:exclude_code_analysis: Suppress MSBuild code analysis for␣
↪patterns
tools.microsoft.msbuildtoolchain:compile_options: Dictionary with MSBuild compiler␣
↪options
tools.system.package_manager:mode: Mode for package_manager tools: 'check' or 'install
↪'
tools.system.package_manager:sudo: Use 'sudo' when invoking the package manager tools␣
↪in Linux (False by default)
tools.system.package_manager:sudo_askpass: Use the '-A' argument if using sudo in␣
↪Linux to invoke the system package manager (False by default)
tools.system.package_manager:tool: Default package manager tool: 'apt-get', 'yum',
↪'dnf', 'brew', 'pacman', 'choco', 'zypper', 'pkg' or 'pkgutil'
```

User/Tools configurations

Tools and user configurations can be defined in both the *global.conf* file and *Conan profiles*. They look like:

Listing 54: *global.conf*

```
tools.microsoft.msbuild:verbosity=Diagnostic
tools.microsoft.msbuild:max_cpu_count=2
tools.microsoft.msbuild:vs_version = 16
tools.build:jobs=10
# User conf variable
user.confvar:something=False
```

Important: Profiles values will have priority over globally defined ones in global.conf.

Configuration file template

It is possible to use **jinja2** template engine for *global.conf*. When Conan loads this file, it immediately parses and renders the template, which must result in a standard tools-configuration text.

```
# Using all the cores automatically
tools.build:jobs={{os.cpu_count()}}
# Using the current OS
user.myconf.system:name = {{platform.system()}}
```

The Python packages passed to render the template are `os` and `platform` for all platforms and `distro` in Linux platforms.

Configuration data types

All the values will be interpreted by Conan as the result of the python built-in *eval()* function:

```
# String
tools.microsoft.msbuild:verbosity=Diagnostic
# Boolean
tools.system.package_manager:sudo=True
# Integer
tools.microsoft.msbuild:max_cpu_count=2
# List of values
user.myconf.build:ldflags=["--flag1", "--flag2"]
# Dictionary
tools.microsoft.msbuildtoolchain:compile_options={"ExceptionHandling": "Async"}
```

Configuration data operators

It's also possible to use some extra operators when you're composing tool configurations in your *global.conf* or any of your profiles:

- `+=` == `append`: appends values at the end of the existing value (only for lists).
- `=+` == `prepend`: puts values at the beginning of the existing value (only for lists).
- `=!` == `unset`: gets rid of any configuration value.

Listing 55: *global.conf*

```
# Define the value => ["-f1"]
user.myconf.build:flags=["-f1"]

# Append the value ["-f2"] => ["-f1", "-f2"]
user.myconf.build:flags+=["-f2"]

# Prepend the value ["-f0"] => ["-f0", "-f1", "-f2"]
user.myconf.build:flags=+["-f0"]

# Unset the value
user.myconf.build:flags=!
```

Configuration patterns

You can use package patterns to apply the configuration in those dependencies which are matching:

```
*:tools.cmake.cmaketoolchain:generator=Ninja
zlib:tools.cmake.cmaketoolchain:generator=Visual Studio 16 2019
```

This example shows you how to specify a general `generator` for all your packages except for *zlib* which is defining *Visual Studio 16 2019* as its generator.

Besides that, it's quite relevant to say that **the order matters**. So, if we change the order of the configuration lines above:

```
zlib:tools.cmake.cmaketoolchain:generator=Visual Studio 16 2019
*:tools.cmake.cmaketoolchain:generator=Ninja
```

The result is that you're specifying a general *generator* for all your packages, and that's it. The *zlib* line has no effect because it's the first one evaluated, and after that, Conan is overriding that specific pattern with the most general one, so it deserves to pay special attention to the order.

Configuration of client certificates

Conan supports client TLS certificates. You can configure the path to your existing *Cacert* file and/or your client certificate (and the key) using the following configuration variables:

- `core.net.http:cacert_path`: Path containing a custom Cacert file.
- `core.net.http:client_cert`: Path or tuple of files containing a client certificate (and the key). See more details in Python requests and Client Side Certificates

For instance:

Listing 56: **[CONAN_HOME]/global.conf**

```
core.net.http:cacert_path=/path/to/cacert.pem
core.net.http:client_cert=('/path/client.cert', '/path/client.key')
```

See also:

- *Managing configuration in your recipes (self.conf_info)*

7.5.2 profiles

Introduction to profiles

Conan profiles allow users to set a complete configuration set for **settings**, **options**, **environment variables** (for build time and runtime context), **tool requirements**, and **configuration variables** in a file.

They have this structure:

```
[settings]
arch=x86_64
build_type=Release
os=Macos

[options]
MyLib:shared=True

[buildenv]
VAR1=value

[tool_requires]
tool1/0.1@user/channel
*: tool4/0.1@user/channel

[conf]
tools.build:jobs=2
```

Profiles can be created with the `detect` option in *conan profile* command, and edited later. If you don't specify a *name*, the command will create the `default` profile:

Listing 57: *Creating the Conan default profile*

```
$ conan profile detect
apple-clang>=13, using the major as version
Detected profile:
[settings]
arch=x86_64
build_type=Release
compiler=apple-clang
compiler.cppstd=gnu17
compiler.libcxx=libc++
compiler.version=14
os=Macos

WARN: This profile is a guess of your environment, please check it.
WARN: Defaulted to cppstd='gnu17' for apple-clang.
WARN: The output of this command is not guaranteed to be stable and can change in
↪future Conan versions.
WARN: Use your own profile files for stability.
Saving detected profile to [CONAN_HOME]/profiles/default
```

Note: A note about the detected C++ standard by Conan

Conan will always set the default C++ standard as the one that the detected compiler version uses by default, except for the case of macOS using apple-clang. In this case, for apple-clang>=11, it sets `compiler.cppstd=gnu17`. If you want to use a different C++ standard, you can edit the default profile file directly.

Listing 58: *Creating another profile: myprofile*

```
$ conan profile detect --name myprofile
Found apple-clang 14.0
apple-clang>=13, using the major as version
Detected profile:
[settings]
arch=x86_64
build_type=Release
compiler=apple-clang
compiler.cppstd=gnu17
compiler.libcxx=libc++
compiler.version=14
os=Macos

WARN: This profile is a guess of your environment, please check it.
WARN: Defaulted to cppstd='gnu17' for apple-clang.
WARN: The output of this command is not guaranteed to be stable and can change in␣
↪future Conan versions.
WARN: Use your own profile files for stability.
Saving detected profile to [CONAN_HOME]/profiles/myprofile
```

Profile files can be used with `-pr/--profile` option in many commands like **conan install** or **conan create** commands. If you don't specify any profile at all, the `default` profile will be always used:

Listing 59: Using the *default* profile

```
$ conan create .
```

Listing 60: Using a *myprofile* profile

```
$ conan create . -pr=myprofile
```

Profiles can be located in different folders:

```
$ conan install . -pr /abs/path/to/myprofile   # abs path
$ conan install . -pr ./relpath/to/myprofile   # resolved to current dir
$ conan install . -pr ../relpath/to/myprofile  # resolved to relative dir
$ conan install . -pr myprofile  # resolved to [CONAN_HOME]/profiles/myprofile
```

Listing existing profiles in the *profiles* folder can be done like this:

```
$ conan profile list
Profiles found in the cache:
default
myprofile1
myprofile2
...
```

You can also show the profile's content per context:

```
$ conan profile show -pr myprofile
Host profile:
[settings]
arch=x86_64
build_type=Release
```

(continues on next page)

(continued from previous page)

```
compiler=apple-clang
compiler.cppstd=gnu17
compiler.libcxx=libc++
compiler.version=14
os=Macos

Build profile:
[settings]
arch=x86_64
build_type=Release
compiler=apple-clang
compiler.cppstd=gnu17
compiler.libcxx=libc++
compiler.version=14
os=Macos
```

See also:

- Manage your profiles and share them using *conan config install*.
- Check the command and its sub-comands of *conan profile*.

Profile sections

These are the available sections in profiles:

[settings]

List of settings available from *settings.yml*:

Listing 61: *myprofile*

```
[settings]
arch=x86_64
build_type=Release
compiler=apple-clang
compiler.cppstd=gnu17
compiler.libcxx=libc++
compiler.version=14
os=Macos
```

[options]

List of options available from your recipe and its dependencies:

Listing 62: *myprofile*

```
[options]
my_pkg_option=True
shared=True
```

[tool_requires]

List of `tool_requires` required by your recipe or its dependencies:

Listing 63: *myprofile*

```
[tool_requires]
cmake/3.25.2
```

See also:

Read more about tool requires in this section: *Using build tools as Conan packages*.

[system_tools]

Warning: This feature is experimental and subject to breaking changes.

This section is similar to the previous one, **[tool_requires]**, but it's intended to list only the tool requires that are already in your own system and you don't want Conan to search for, neither remotely nor locally.

For instance, you have already installed `cmake==3.24.2` in your system:

```
$ cmake --version
cmake version 3.24.2

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

Now, you have in your recipe (or the transitive dependencies) declared a **tool_requires**, i.e., something like this:

Listing 64: **conanfile.py**

```
from conan import ConanFile

class PkgConan(ConanFile):
    name = "pkg"
    version = "2.0"
    # ....

    # Exact version
    def build_requirements(self):
        self.tool_requires("cmake/3.24.2")

    # Or even version ranges
    def build_requirements(self):
        self.tool_requires("cmake/[>=3.20.0]")
```

Given this situation, it could make sense to want to use your already installed CMake version, so it's enough to declare it as a `system_tools` in your profile (`default` one or any other in use):

Listing 65: *myprofile*

```
...

[system_tools]
cmake/3.24.2
```

Whenever you want to create the package, you'll see that build requirement is already satisfied because of the system tool declaration:

```
$ conan create . -pr myprofile --build=missing
...
-------- Computing dependency graph --------
Graph root
    virtual
Requirements
    pkg/2.0#3488ec5c2829b44387152a6c4b013767 - Cache
Build requirements
    cmake/3.24.2 - System tool

-------- Computing necessary packages --------

-------- Computing necessary packages --------
pkg/2.0: Forced build from source
Requirements
    pkg/2.0#3488ec5c2829b44387152a6c4b013767:20496b332552131b67fb99bf425f95f64d0d0818
↪- Build
Build requirements
    cmake/3.24.2 - System tool
```

Notice that if the `system_tools` declared does not make a strict match with the `tool_requires` one (version or version range), then Conan will try to bring them remotely or locally as usual. Given the previous example, changing the profile as follows:

Listing 66: *myprofile*

```
...

[system_tools]
cmake/3.20.0
```

The result will be different when calling the **conan create**, because Conan will download remotely and build from source if necessary:

```
$ conan create . -pr myprofile --build=missing
...
-------- Computing dependency graph --------
Graph root
    virtual
Requirements
    pkg/2.0#3488ec5c2829b44387152a6c4b013767 - Cache
Build requirements
    cmake/3.24.2#e35bc44b3fcbcd661e0af0dc5b5b1ad4 - Downloaded (conancenter)

-------- Computing necessary packages --------

-------- Computing necessary packages --------
pkg/2.0: Forced build from source
Requirements
    pkg/2.0#3488ec5c2829b44387152a6c4b013767:20496b332552131b67fb99bf425f95f64d0d0818
↪- Build
Build requirements
    cmake/3.24.2
↪#e35bc44b3fcbcd661e0af0dc5b5b1ad4:d0599452a426a161e02a297c6e0c5070f99b4909 - Build
```

[buildenv]

List of environment variables that will be injected to the environment every time the ConanFile `run(cmd, env="conanbuild")` method is invoked (build time context is automatically run by *VirtualBuildEnv*).

Besides that, it is able to apply some additional operators to each variable declared when you're composing profiles or even local variables:

- `+=` == `append`: appends values at the end of the existing value.
- `=+` == `prepend`: puts values at the beginning of the existing value.
- `=!` == `unset`: gets rid of any variable value.

Another essential point to mention is the possibility of defining variables as *PATH* ones by simply putting `(path)` as the prefix of the variable. It is useful to automatically get the append/prepend of the *PATH* in different systems (Windows uses `;` as separation, and UNIX `:`).

Listing 67: *myprofile*

```
[buildenv]
# Define a variable "MyVar1"
MyVar1=My Value; other

# Append another value to "MyVar1"
MyVar1+=MyValue12

# Define a PATH variable "MyPath1"
MyPath1=(path)/some/path11

# Prepend another PATH to "MyPath1"
MyPath1=+(path)/other path/path12

# Unset the variable "MyPath1"
MyPath1=!
```

Then, the result of applying this profile is:

- `MyVar1`: `My Value; other MyValue12`
- **`MyPath1`:**
  - Unix: `/other path/path12:/some/path11`
  - Windows: `/other path/path12;/some/path11`
- `mypkg*:PATH`: `None`

[runenv]

List of environment variables that will be injected to the environment every time the ConanFile `run(cmd, env="conanrun")` method is invoked (runtime context is automatically run by *VirtualRunEnv*).

All the operators/patterns explained for *[buildenv]* applies to this one in the same way:

Listing 68: *myprofile*

```
[runenv]
MyVar1=My Value; other
MyVar1+=MyValue12
```

(continues on next page)

(continued from previous page)

```
MyPath1=(path)/some/path11
MyPath1=+(path)/other path/path12
MyPath1=!
```

[conf]

Note: It's recommended to have previously read the *global.conf* section.

List of user/tools configurations:

Listing 69: *myprofile*

```
[conf]
tools.microsoft.msbuild:verbosity=Diagnostic
tools.microsoft.msbuild:max_cpu_count=2
tools.microsoft.msbuild:vs_version = 16
tools.build:jobs=10
# User conf variable
user.confvar:something=False
```

They can also be used in *global.conf*, but **profiles values will have priority over globally defined ones in global.conf**, so let's see an example that is a bit more complex, trying different configurations coming from the *global.conf* and another profile *myprofile*:

Listing 70: *global.conf*

```
# Defining several lists
user.myconf.build:ldflags=["--flag1 value1"]
user.myconf.build:cflags=["--flag1 value1"]
```

Listing 71: *myprofile*

```
[settings]
...

[conf]
# Appending values into the existing list
user.myconf.build:ldflags+=["--flag2 value2"]

# Unsetting the existing value (it'd be like we define it as an empty value)
user.myconf.build:cflags=!

# Prepending values into the existing list
user.myconf.build:ldflags=+["--prefix prefix-value"]
```

Running, for instance, **conan install . -pr myprofile**, the configuration output will be something like:

```
...
Configuration:
[settings]
[options]
[tool_requires]
```

(continues on next page)

(continued from previous page)

```
[conf]
user.myconf.build:cflags=!
user.myconf.build:ldflags=['--prefix prefix-value', '--flag1 value1', '--flag2 value2
↪']
...
```

Profile rendering

The profiles are rendered as **jinja2** templates by default. When Conan loads a profile, it immediately parses and renders the template, which must result in a standard text profile.

Some of the capabilities of the profile templates are:

- Using the platform information, like obtaining the current OS, is possible because the Python `platform` module is added to the render context:

  Listing 72: *profile_vars*

  ```
  [settings]
  os = {{ {"Darwin": "Macos"}.get(platform.system(), platform.system()) }}
  ```

- Reading environment variables can be done because the Python `os` module is added to the render context:

  Listing 73: *profile_vars*

  ```
  [settings]
  build_type = {{ os.getenv("MY_BUILD_TYPE") }}
  ```

- Defining your own variables and using them in the profile:

  Listing 74: *profile_vars*

  ```
  {% set os = "FreeBSD" %}
  {% set clang = "my/path/to/clang" %}

  [settings]
  os = {{ os }}

  [conf]
  tools.build:compiler_executables={'c': '{{ clang }}', 'cpp': '{{ clang + '++' }}'␣
  ↪}
  ```

- Joining and defining paths, including referencing the current profile directory. For example, defining a toolchain whose file is located besides the profile can be done. Besides the `os` Python module, the variable `profile_dir` pointing to the current profile folder is added to the context.

  Listing 75: *profile_vars*

  ```
  [conf]
  tools.cmake.cmaketoolchain:toolchain_file = {{ os.path.join(profile_dir,
  ↪"toolchain.cmake") }}
  ```

- Including or importing other files from `profiles` folder:

Listing 76: *profile_vars*

```
{% set a = "Debug" %}
```

Listing 77: *myprofile*

```
{% import "profile_vars" as vars %}
[settings]
build_type = {{ vars.a }}
```

- Any other feature supported by *jinja2* is possible: for loops, if-else, etc. This would be useful to define custom per-package settings or options for multiple packages in a large dependency graph.

Profile patterns

Profiles also support patterns definition, so you can override some settings, configuration variables, etc. for some specific packages:

Listing 78: *zlib_clang_profile*

```
[settings]
# Only for zlib
zlib*:compiler=clang
zlib*:compiler.version=3.5
zlib*:compiler.libcxx=libstdc++11

# For the all the dependency tree
compiler=gcc
compiler.version=4.9
compiler.libcxx=libstdc++11

[options]
# shared=True option only for zlib package
zlib*:shared=True

[buildenv]
# For the all the dependency tree
*:MYVAR=my_var

[conf]
# Only for zlib
zlib*:tools.build:compiler_executables={'c': '/usr/bin/clang', 'cpp': '/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.

Important: Putting only `zlib:` is not going to work, you have to always put a pattern-like expression, e.g., `zlib*:`, `zlib/1.*:`, etc.

They accept patterns too, like `-s *@myuser/*`, which means that packages that have the username "myuser" will use clang 3.5 as compiler, and gcc otherwise:

Listing 79: *myprofile*

```
[settings]
*@myuser/*:compiler=clang
*@myuser/*:compiler.version=3.5
*@myuser/*:compiler.libcxx=libstdc++11
compiler=gcc
compiler.version=4.9
compiler.libcxx=libstdc++11
```

Also *&* can be specified as the package name. It will apply only to the consumer conanfile (.py or .txt). This is a special case because the consumer conanfile might not declare a *name* so it would be impossible to reference it.

Listing 80: *myprofile*

```
[settings]
&:compiler=gcc
&:compiler.version=4.9
&:compiler.libcxx=libstdc++11
```

Profile includes

You can include other profile files 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 81: *gcc_49*

```
[settings]
compiler=gcc
compiler.version=4.9
compiler.libcxx=libstdc++11
```

Listing 82: *myprofile*

```
include(gcc_49)

[settings]
zlib*:compiler=clang
zlib*:compiler.version=3.5
zlib*:compiler.libcxx=libstdc++11
```

The final result of using *myprofile* is:

Listing 83: *myprofile (virtual result)*

```
[settings]
compiler=gcc
compiler.libcxx=libstdc++11
compiler.version=4.9
zlib*:compiler=clang
zlib*:compiler.libcxx=libstdc++11
zlib*:compiler.version=3.5
```

See also:

- *How to compose two or more profiles*

7.5.3 settings.yml

This configuration file is located in the Conan user home, i.e., `[CONAN_HOME]/settings.yml`. It looks like this:

```
# This file was generated by Conan. Remove this comment if you edit this file or Conan
# will destroy your changes.
os:
    Windows:
        subsystem: [null, cygwin, msys, msys2, wsl]
    WindowsStore:
        version: ["8.1", "10.0"]
    WindowsCE:
        platform: ANY
        version: ["5.0", "6.0", "7.0", "8.0"]
    Linux:
    iOS:
        version: &ios_version
                      ["7.0", "7.1", "8.0", "8.1", "8.2", "8.3", "9.0", "9.1", "9.2", "9.
↪3", "10.0", "10.1", "10.2", "10.3",
                       "11.0", "11.1", "11.2", "11.3", "11.4", "12.0", "12.1", "12.2",
↪"12.3", "12.4",
                       "13.0", "13.1", "13.2", "13.3", "13.4", "13.5", "13.6", "13.7",
                       "14.0", "14.1", "14.2", "14.3", "14.4", "14.5", "14.6", "14.7",
↪"14.8",
                       "15.0", "15.1", "15.2", "15.3", "15.4", "15.5", "15.6", "16.0",
↪"16.1"]
        sdk: ["iphoneos", "iphonesimulator"]
        sdk_version: [null, "11.3", "11.4", "12.0", "12.1", "12.2", "12.4",
                           "13.0", "13.1", "13.2", "13.4", "13.5", "13.6", "13.7",
                           "14.0", "14.1", "14.2", "14.3", "14.4", "14.5", "15.0", "15.2
↪", "15.4", "15.5", "16.0", "16.1"]
    watchOS:
        version: ["4.0", "4.1", "4.2", "4.3", "5.0", "5.1", "5.2", "5.3", "6.0", "6.1
↪", "6.2",
                       "7.0", "7.1", "7.2", "7.3", "7.4", "7.5", "7.6", "8.0", "8.1", "8.
↪3", "8.4", "8.5", "8.6", "8.7", "9.0", "9.1"]
        sdk: ["watchos", "watchsimulator"]
        sdk_version: [null, "4.3", "5.0", "5.1", "5.2", "5.3", "6.0", "6.1", "6.2",
                           "7.0", "7.1", "7.2", "7.4", "8.0", "8.0.1", "8.3", "8.5", "9.0
↪", "9.1"]
    tvOS:
        version: ["11.0", "11.1", "11.2", "11.3", "11.4", "12.0", "12.1", "12.2", "12.
↪3", "12.4",
                       "13.0", "13.2", "13.3", "13.4", "14.0", "14.2", "14.3", "14.4",
↪"14.5", "14.6", "14.7",
                       "15.0", "15.1", "15.2", "15.3", "15.4", "15.5", "15.6", "16.0",
↪"16.1"]
        sdk: ["appletvos", "appletvsimulator"]
        sdk_version: [null, "11.3", "11.4", "12.0", "12.1", "12.2", "12.4",
                           "13.0", "13.1", "13.2", "13.4", "14.0", "14.2", "14.3", "14.5
↪", "15.0", "15.2", "15.4", "16.0", "16.1"]
    Macos:
        version: [null, "10.6", "10.7", "10.8", "10.9", "10.10", "10.11", "10.12",
↪"10.13", "10.14", "10.15", "11.0", "12.0", "13.0"]
        sdk_version: [null, "10.13", "10.14", "10.15", "11.0", "11.1", "11.3", "12.0",
↪ "12.1", "12.3", "13.0"]
```

(continues on next page)

(continued from previous page)

```
        subsystem:
            null:
            catalyst:
                ios_version: *ios_version
    Android:
        api_level: [ANY]
    FreeBSD:
    SunOS:
    AIX:
    Arduino:
        board: [ANY]
    Emscripten:
    Neutrino:
        version: ["6.4", "6.5", "6.6", "7.0", "7.1"]
    baremetal:
    VxWorks:
        version: ["7"]
arch: [x86, x86_64, ppc32be, ppc32, ppc64le, ppc64,
       armv4, armv4i, armv5el, armv5hf, armv6, armv7, armv7hf, armv7s, armv7k, armv8,␣
↪armv8_32, armv8.3,
       sparc, sparcv9,
       mips, mips64, avr, s390, s390x, asm.js, wasm, sh4le,
       e2k-v2, e2k-v3, e2k-v4, e2k-v5, e2k-v6, e2k-v7,
       xtensalx6, xtensalx106, xtensalx7]
compiler:
    sun-cc:
        version: ["5.10", "5.11", "5.12", "5.13", "5.14", "5.15"]
        threads: [null, posix]
        libcxx: [libCstd, libstdcxx, libstlport, libstdc++]
    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", "5.5",
                  "6", "6.1", "6.2", "6.3", "6.4", "6.5",
                  "7", "7.1", "7.2", "7.3", "7.4", "7.5",
                  "8", "8.1", "8.2", "8.3", "8.4", "8.5",
                  "9", "9.1", "9.2", "9.3", "9.4", "9.5",
                  "10", "10.1", "10.2", "10.3", "10.4",
                  "11", "11.1", "11.2", "11.3",
                  "12", "12.1", "12.2"]
        libcxx: [libstdc++, libstdc++11]
        threads: [null, posix, win32]  # Windows MinGW
        exception: [null, dwarf2, sjlj, seh]  # Windows MinGW
        cppstd: [null, 98, gnu98, 11, gnu11, 14, gnu14, 17, gnu17, 20, gnu20, 23,␣
↪gnu23]
    msvc:
        version: [170, 180, 190, 191, 192, 193]
        update: [null, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
        runtime: [static, dynamic]
        runtime_type: [Debug, Release]
        cppstd: [null, 14, 17, 20, 23]
        toolset: [null, v110_xp, v120_xp, v140_xp, v141_xp]
    clang:
        version: ["3.3", "3.4", "3.5", "3.6", "3.7", "3.8", "3.9", "4.0",
                  "5.0", "6.0", "7.0", "7.1",
                  "8", "9", "10", "11", "12", "13", "14", "15", "16"]
        libcxx: [null, libstdc++, libstdc++11, libc++, c++_shared, c++_static]
        cppstd: [null, 98, gnu98, 11, gnu11, 14, gnu14, 17, gnu17, 20, gnu20, 23,␣
↪gnu23]
```

(continues on next page)

(continued from previous page)

```
        runtime: [null, static, dynamic]
        runtime_type: [null, Debug, Release]
        runtime_version: [null, v140, v141, v142, v143]
    apple-clang:
        version: ["5.0", "5.1", "6.0", "6.1", "7.0", "7.3", "8.0", "8.1", "9.0", "9.1
↪", "10.0", "11.0", "12.0", "13", "13.0", "13.1", "14", "14.0"]
        libcxx: [libstdc++, libc++]
        cppstd: [null, 98, gnu98, 11, gnu11, 14, gnu14, 17, gnu17, 20, gnu20, 23,␣
↪gnu23]
    intel-cc:
        version: ["2021.1", "2021.2", "2021.3"]
        update: [null, ANY]
        mode: ["icx", "classic", "dpcpp"]
        libcxx: [null, libstdc++, libstdc++11, libc++]
        cppstd: [null, 98, gnu98, 03, gnu03, 11, gnu11, 14, gnu14, 17, gnu17, 20,␣
↪gnu20, 23, gnu23]
        runtime: [null, static, dynamic]
        runtime_type: [null, Debug, Release]
    qcc:
        version: ["4.4", "5.4", "8.3"]
        libcxx: [cxx, gpp, cpp, cpp-ne, accp, acpp-ne, ecpp, ecpp-ne]
        cppstd: [null, 98, gnu98, 11, gnu11, 14, gnu14, 17, gnu17]
    mcst-lcc:
        version: ["1.19", "1.20", "1.21", "1.22", "1.23", "1.24", "1.25"]
        libcxx: [libstdc++, libstdc++11]
        cppstd: [null, 98, gnu98, 11, gnu11, 14, gnu14, 17, gnu17, 20, gnu20, 23,␣
↪gnu23]

build_type: [null, Debug, Release, RelWithDebInfo, MinSizeRel]
```

As you can see, the possible values of `settings` are defined in the same file. This is done to ensure matching naming and spelling as well as defining a common settings model among users and the OSS community. Some general information about settings:

- If a setting is allowed to be set to any value, you can use `ANY`.
- If a setting is allowed to be set to any value or it can also be unset, you can use `[null, ANY]`.

However, this configuration file can be modified to any needs, including new settings or sub-settings and their values. If you want to distribute an unified *settings.yml* file you can use the *conan config install command*.

See also:

- *Conan packages binary compatibility: the package ID*

Operating systems

`baremetal` operating system is a convention meaning that the binaries run directly on the hardware, without an operating system or equivalent layer. This is to differentiate to the `null` value, which is associated to the "this value is not defined" semantics. `baremetal` is a common name convention for embedded microprocessors and microcontrollers' code. It is expected that users might customize the space inside the `baremetal` setting with further subsettings to specify their specific hardware platforms, boards, families, etc. At the moment the `os=baremetal` value is still not used by Conan builtin toolchains and helpers, but it is expected that they can evolve and start using it.

Compilers

Some notes about different compilers:

msvc

- It uses the compiler version, that is 190 (19.0), 191 (19.1), etc, instead of the Visual Studio IDE (15, 16, etc).
- It is only used by the new build integrations in *conan.tools.cmake* and *conan.tools.microsoft*, but not the previous ones.

When using the `msvc` compiler, the Visual Studio toolset version (the actual `vcvars` activation and `MSBuild` location) will be defined by the default provided by that compiler version:

- `msvc` compiler version '190': Visual Studio 14 2015
- `msvc` compiler version '191': Visual Studio 15 2017
- `msvc` compiler version '192': Visual Studio 16 2019
- `msvc` compiler version '193': Visual Studio 17 2022

This can be configured in your profiles with the `tools.microsoft.msbuild:vs_version` configuration:

```
[settings]
compiler=msvc
compiler.version=190

[conf]
tools.microsoft.msbuild:vs_version = 16
```

In this case, the `vcvars` will activate the Visual Studio 16 installation, but the `190` compiler version will still be used because the necessary `toolset=v140` will be set.

The settings define the last digit `update:  [null, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]`, which by default is `null` and means that Conan assumes binary compatibility for the compiler patches, which works in general for the Microsoft compilers. For cases where finer control is desired, you can just add the `update` part to your profiles:

```
[settings]
compiler=msvc
compiler.version=191
compiler.version.update=3
```

This will be equivalent to the full version `1913 (19.13)`. If even further details are desired, you could even add your own digits to the `update` subsetting in `settings.yml`.

intel-cc

> **Warning:** This feature is experimental and subject to breaking changes.

This compiler is aimed to handle the new Intel oneAPI DPC++/C++/Classic compilers. Instead of having *n* different compilers, you have 3 different **modes** of working:

- `icx` for Intel oneAPI C++.
- `dpcpp` for Intel oneAPI DPC++.

- `classic` for Intel C++ Classic ones.

Besides that, Intel releases some versions with revisions numbers so the `update` field is supposed to be any possible minor number for the Intel compiler version used, e.g, `compiler.version=2021.1` and `compiler.update=311` mean Intel version is `2021.1.311`.

Architectures

Here you can find a brief explanation of each of the architectures defined as `arch`, `arch_build` and `arch_target` settings.

- **x86**: The popular 32 bit x86 architecture.
- **x86_64**: The popular 64 bit x64 architecture.
- **ppc64le**: The PowerPC 64 bit Big Endian architecture.
- **ppc32**: The PowerPC 32 bit architecture.
- **ppc64le**: The PowerPC 64 bit Little Endian architecture.
- **ppc64**: The PowerPC 64 bit Big Endian architecture.
- **armv5el**: The ARM 32 bit version 5 architecture, soft-float.
- **armv5hf**: The ARM 32 bit version 5 architecture, hard-float.
- **armv6**: The ARM 32 bit version 6 architecture.
- **armv7**: The ARM 32 bit version 7 architecture.
- **armv7hf**: The ARM 32 bit version 7 hard-float architecture.
- **armv7s**: The ARM 32 bit version 7 *swift* architecture mostly used in Apple's A6 and A6X chips on iPhone 5, iPhone 5C and iPad 4.
- **armv7k**: The ARM 32 bit version 7 *k* architecture mostly used in Apple's WatchOS.
- **armv8**: The ARM 64 bit and 32 bit compatible version 8 architecture. It covers only the `aarch64` instruction set.
- **armv8_32**: The ARM 32 bit version 8 architecture. It covers only the `aarch32` instruction set (a.k.a. `ILP32`).
- **armv8.3**: The ARM 64 bit and 32 bit compatible version 8.3 architecture. Also known as `arm64e`, it is used on the A12 chipset added in the latest iPhone models (XS/XS Max/XR).
- **sparc**: The SPARC (Scalable Processor Architecture) originally developed by Sun Microsystems.
- **sparcv9**: The SPARC version 9 architecture.
- **mips**: The 32 bit MIPS (Microprocessor without Interlocked Pipelined Stages) developed by MIPS Technologies (formerly MIPS Computer Systems).
- **mips64**: The 64 bit MIPS (Microprocessor without Interlocked Pipelined Stages) developed by MIPS Technologies (formerly MIPS Computer Systems).
- **avr**: The 8 bit AVR microcontroller architecture developed by Atmel (Microchip Technology).
- **s390**: The 32 bit address Enterprise Systems Architecture 390 from IBM.
- **s390x**: The 64 bit address Enterprise Systems Architecture 390 from IBM.
- **asm.js**: The subset of JavaScript that can be used as low-level target for compilers, not really a processor architecture, it's produced by Emscripten. Conan treats it as an architecture to align with build systems design (e.g. GNU auto tools and CMake).

- **wasm**: The Web Assembly, not really a processor architecture, but byte-code format for Web, it's produced by Emscripten. Conan treats it as an architecture to align with build systems design (e.g. GNU auto tools and CMake).
- **sh4le**: The Hitachi SH-4 SuperH architecture.
- **e2k-v2**: The Elbrus 2000 v2 512 bit VLIW (Very Long Instruction Word) architecture (Elbrus 2CM, Elbrus 2C+ CPUs) originally developed by MCST (Moscow Center of SPARC Technologies).
- **e2k-v3**: The Elbrus 2000 v3 512 bit VLIW (Very Long Instruction Word) architecture (Elbrus 2S, aka Elbrus 4C, CPU) originally developed by MCST (Moscow Center of SPARC Technologies).
- **e2k-v4**: The Elbrus 2000 v4 512 bit VLIW (Very Long Instruction Word) architecture (Elbrus 8C, Elbrus 8C1, Elbrus 1C+ and Elbrus 1CK CPUs) originally developed by MCST (Moscow Center of SPARC Technologies).
- **e2k-v5**: The Elbrus 2000 v5 512 bit VLIW (Very Long Instruction Word) architecture (Elbrus 8C2 ,aka Elbrus 8CB, CPU) originally developed by MCST (Moscow Center of SPARC Technologies).
- **e2k-v6**: The Elbrus 2000 v6 512 bit VLIW (Very Long Instruction Word) architecture (Elbrus 2C3, Elbrus 12C and Elbrus 16C CPUs) originally developed by MCST (Moscow Center of SPARC Technologies).
- **e2k-v7**: The Elbrus 2000 v7 512 bit VLIW (Very Long Instruction Word) architecture (Elbrus 32C CPU) originally developed by MCST (Moscow Center of SPARC Technologies).
- **xtensalx6**: Xtensa LX6 DPU for ESP32 microcontroller.
- **xtensalx106**: Xtensa LX6 DPU for ESP8266 microcontroller.
- **xtensalx7**: Xtensa LX7 DPU for ESP32-S2 and ESP32-S3 microcontrollers.

C++ standard libraries (aka compiler.libcxx)

`compiler.libcxx` sub-setting defines C++ standard libraries implementation to be used. The sub-setting applies only to certain compilers, e.g. it applies to *clang*, *apple-clang* and *gcc*, but doesn't apply to *Visual Studio*.

- **libstdc++** (gcc, clang, apple-clang, sun-cc): The GNU C++ Library. NOTE that this implicitly defines **_GLIBCXX_USE_CXX11_ABI=0** to use old ABI. Might be a wise choice for old systems, such as CentOS 6. On Linux systems, you may need to install libstdc++-dev (package name could be different in various distros) in order to use the standard library. NOTE that on Apple systems usage of **libstdc++** has been deprecated.
- **libstdc++11** (gcc, clang, apple-clang): The GNU C++ Library. NOTE that this implicitly defines **_GLIBCXX_USE_CXX11_ABI=1** to use new ABI. Might be a wise choice for newer systems, such as Ubuntu 20. On Linux systems, you may need to install libstdc++-dev (package name could be different in various distros) in order to use the standard library. NOTE that on Apple systems usage of **libstdc++** has been deprecated.
- **libc++** (clang, apple-clang): LLVM libc++. On Linux systems, you may need to install libc++-dev (package name could be different in various distros) in order to use the standard library.
- **c++_shared** (clang, Android only): use LLVM libc++ as a shared library. Refer to the C++ Library Support for the additional details.
- **c++_static** (clang, Android only): use LLVM libc++ as a static library. Refer to the C++ Library Support for the additional details.
- **libCstd** (sun-cc): Rogue Wave's stdlib. See Comparing C++ Standard Libraries libCstd, libstlport, and libstdcxx.
- **libstlport** (sun-cc): STLport. See Comparing C++ Standard Libraries libCstd, libstlport, and libstdcxx.
- **libstdcxx** (sun-cc): Apache C++ Standard Library. See Comparing C++ Standard Libraries libCstd, libstlport, and libstdcxx.
- **gpp** (qcc): GNU C++ lib. See QCC documentation.

- **cpp** (qcc): Dinkum C++ lib. See QCC documentation.
- **cpp-ne** (qcc): Dinkum C++ lib (no exceptions). See QCC documentation.
- **acpp** (qcc): Dinkum Abridged C++ lib. See QCC documentation.
- **acpp-ne** (qcc): Dinkum Abridged C++ lib (no exceptions). See QCC documentation.
- **ecpp** (qcc): Embedded Dinkum C++ lib. See QCC documentation.
- **ecpp-ne** (qcc): Embedded Dinkum C++ lib (no exceptions). See QCC documentation.
- **cxx** (qcc): LLVM C++. See QCC documentation.

Customizing settings

Settings are also customizable to add your own ones:

Adding new settings

It is possible to add new settings at the root of the *settings.yml* file, something like:

```
os:
    Windows:
        subsystem: [null, cygwin, msys, msys2, wsl]
distro: [null, RHEL6, CentOS, Debian]
```

If we want to create different binaries from our recipes defining this new setting, we would need to add to our recipes that:

```
class Pkg(ConanFile):
    settings = "os", "compiler", "build_type", "arch", "distro"
```

The value `null` allows for not defining it (which would be a default value, valid for all the other distros). It is also possible to define values for it in the profiles:

```
[settings]
os = "Linux"
distro = "CentOS"
compiler = "gcc"
```

And use their values to affect our build if desired:

```
class Pkg(ConanFile):
    settings = "os", "compiler", "build_type", "arch", "distro"

    def generate(self):
        tc = CMakeToolchain(self)
        if self.settings.distro == "CentOS":
            tc.cache_variables["SOME_CENTOS_FLAG"] = "Some CentOS Value"
            ...
```

Adding new sub-settings

The above approach requires modification to all recipes to take it into account. It is also possible to define kind of incompatible settings, like `os=Windows` and `distro=CentOS`. While adding new settings is totally suitable, it might make more sense to add it as a new sub-setting of the `Linux` OS:

```
os:
    Windows:
        subsystem: [null, cygwin, msys, msys2, wsl]
    Linux:
        distro: [null, RHEL6, CentOS, Debian]
```

With this definition we could define our profiles as:

```
[settings]
os = "Linux"
os.distro = "CentOS"
compiler = "gcc"
```

And any attempt to define `os.distro` for another `os` value rather than `Linux` will raise an error.

As this is a sub-setting, it will be automatically taken into account in all recipes that declare an `os` setting. Note that having a value of `distro=null` possible is important if you want to keep previously created binaries, otherwise you would be forcing to always define a specific distro value, and binaries created without this sub-setting, won't be usable anymore.

The sub-setting can also be accessed from recipes:

```
class Pkg(ConanFile):
    settings = "os", "compiler", "build_type", "arch"  # Note, no "distro" defined␣
↪here

    def generate(self):
        tc = CMakeToolchain(self)
        if self.settings.os == "Linux" and self.settings.os.distro == "CentOS":
            tc.cache_variables["SOME_CENTOS_FLAG"] = "Some CentOS Value"
```

Add new values

In the same way we have added a new `distro` sub-setting, it is possible to add new values to existing settings and sub-settings. For example, if some compiler version is not present in the range of accepted values, you can add those new values.

You can also add a completely new compiler:

```
os:
    Windows:
        subsystem: [null, cygwin, msys, msys2, wsl]
   ...
compiler:
    gcc:
        ...
    mycompiler:
        version: [1.1, 1.2]
    msvc:
```

This works as the above regarding profiles, and the way they can be accessed from recipes. The main issue with custom compilers is that the builtin build helpers, like `CMake`, `MSBuild`, etc, internally contains code that will check for those values. For example, the `MSBuild` build helper will only know how to manage the `msvc` setting and sub-settings, but not the new compiler. For those cases, custom logic can be implemented in the recipes:

```
class Pkg(ConanFile):
    settings = "os", "compiler", "build_type", "arch"

    def build(self):
        if self.settings.compiler == "mycompiler":
            my_custom_compile = ["some", "--flags", "for", "--my=compiler"]
            self.run(["mycompiler", "."] + my_custom_compile)
```

Note: You can remove items from *settings.yml* file: compilers, OS, architectures, etc. Do that only in the case you really want to protect against creation of binaries for other platforms other than your main supported ones. In the general case, you can leave them, the binary configurations are managed in **profiles**, and you want to define your supported configurations in profiles, not by restricting the *settings.yml*

Note: If you customize your *settings.yml*, you can share, distribute and sync this configuration with your team and CI machines with the *conan config install* command.

settings_user.yml

The previous section explains how to customize the Conan *settings.yml*, but you could also create your *settings_user.yml*. This file will contain only the new fields-values that you want to use in your recipes, so the final result will be a composition of both files, the *settings.yml* and the *settings_user.yml*.

See also:

- *Customize your settings: create your settings_user.yml*

7.5.4 remotes.json

The **remotes.json** file is located in the Conan user home directory, e.g., *[CONAN_HOME]/remotes.json*.

The default file created by Conan looks like this:

Listing 84: **remotes.json**

```
{
 "remotes": [
  {
   "name": "conancenter",
   "url": "https://center.conan.io",
   "verify_ssl": true
  }
 ]
}
```

Essentially, it tells Conan where to list/upload/download the recipes/binaries from the remotes specified by their URLs.

The fields for each remote are:

- `name` (Required, `string` value): Name of the remote. This name will be used in commands like *conan list*, e.g., **`conan list zlib/1.2.11 --remote my_remote_name`**.
- `url` (Required, `string` value): indicates the URL to be used by Conan to search for the recipes/binaries.
- `verify_ssl` (Required, `bool` value): Verify SSL certificate of the specified url.

- disabled (Optional, bool value, false by default): If the remote is enabled or not to be used by commands like search, list, download and upload. Notice that a disabled remote can be used to authenticate against it even if it's disabled.

See also:

- *How to manage SSL (TLS) certificates*
- *How to manage remotes.json through CLI: conan remotes*.

7.6 Extensions

Conan can be extended in a few ways, with custom user code:

- python_requires allow to put common recipe code in a recipe package that can be reused by other recipes by declaring a python_requires = "mypythoncode/version"
- hooks are "pre" and "post" recipe methods (like pre_build() and post_build()) extensions that can be used to complement recipes with orthogonal functionality, like quality checks, binary analyzing, logging, etc.
- Binary compatibility compatibility.py extension allows to write custom rules for defining custom binary compatibility accross different settings and options
- The cmd_wrapper.py extension allows to inject arbitrary command wrappers to any self.run() recipe command invocation, which can be useful to inject wrappers as parallelization tools
- The package signing extension allows to sign and verify packages at upload and install time respectively
- Deployers, a mechanism to facilitate copying files from one folder, usually the Conan cache, to user folders

---

Note: Besides the built-in Conan extensions listed in this document, there is a repository that contains extensions for Conan, such as custom commands and deployers, useful for different purposes like artifactory tasks, Conan Center Index, etc.

You can find more information on how to use those extensions in the GitHub repository.

Contents:

7.6.1 Python requires

Introduction

The python_requires feature is a very convenient way to share files and code between different recipes. A python require is a special recipe that does not create packages and it is just intended to be reused by other recipes.

A very simple recipe that we want to reuse could be:

```
from conan import ConanFile

myvar = 123

def myfunct():
    return 234

class Pkg(ConanFile):
    name = "pyreq"
```

(continues on next page)

(continued from previous page)

```
    version = "0.1"
    package_type = "python-require"
```

And then we will make it available to other packages with `conan create .`. Note that a `python-require` package does not create binaries, it is just the recipe part.

```
$ conan create .
# It will only export the recipe, but will NOT create binaries
# python-requires do NOT have binaries
```

We can reuse the above recipe functionality declaring the dependency in the `python_requires` attribute and we can access its members using `self.python_requires["<name>"].module`:

```
from conan import ConanFile

class Pkg(ConanFile):
    name = "pkg"
    version = "0.1"
    python_requires = "pyreq/0.1"

    def build(self):
        v = self.python_requires["pyreq"].module.myvar  # v will be 123
        f = self.python_requires["pyreq"].module.myfunct()  # f will be 234
        self.output.info(f"{v}, {f}")
```

```
$ conan create .
...
pkg/0.1: 123, 234
```

Python requires can also use version ranges, and this can be recommended in many cases if those `python-requires` need to evolve over time:

```
from conan import ConanFile

class Pkg(ConanFile):
    python_requires = "pyreq/[>=1.0 <2]"
```

It is also possible to require more than 1 `python-requires` with `python_requires = "pyreq/0.1", "other/1.2"`

Extending base classes

A common use case would be to declare a base class with methods we want to reuse in several recipes via inheritance. We'd write this base class in a python-requires package:

```
from conan import ConanFile

class MyBase:
    def source(self):
        self.output.info("My cool source!")
    def build(self):
        self.output.info("My cool build!")
    def package(self):
        self.output.info("My cool package!")
```

(continues on next page)

(continued from previous page)

```
    def package_info(self):
        self.output.info("My cool package_info!")

class PyReq(ConanFile):
    name = "pyreq"
    version = "0.1"
    package_type = "python-require"
```

And make it available for reuse with:

```
$ conan create .
```

Note that there are two classes in the recipe file:

- `MyBase` is the one intended for inheritance and doesn't extend `ConanFile`.
- `PyReq` is the one that defines the current package being exported, it is the recipe for the reference `pyreq/0.1`.

Once the package with the base class we want to reuse is available we can use it in other recipes to inherit the functionality from that base class. We'd need to declare the `python_requires` as we did before and we'd need to tell Conan the base classes to use in the attribute `python_requires_extend`. Here our recipe will inherit from the class `MyBase`:

```
from conan import ConanFile

class Pkg(ConanFile):
    name = "pkg"
    version = "0.1"
    python_requires = "pyreq/0.1"
    python_requires_extend = "pyreq.MyBase"
```

The resulting inheritance is equivalent to declare our `Pkg` class as `class Pkg(pyreq.MyBase, ConanFile)`. So creating the package we can see how the methods from the base class are reused:

```
$ conan create .
...
pkg/0.1: My cool source!
pkg/0.1: My cool build!
pkg/0.1: My cool package!
pkg/0.1: My cool package_info!
...
```

In general, base class attributes are not inherited, and should be avoided as much as possible. There are method alternatives to some of them like `export()` or `set_version()`. For exceptional situations, see the `init()` method documentation for more information to extend inherited attributes.

Reusing files

It is possible to access the files exported by a recipe that is used with `python_requires`. We could have this recipe, together with a *myfile.txt* file containing the "Hello" text.

```
from conan import ConanFile

class PyReq(ConanFile):
    name = "pyreq"
    version = "1.0"
```

(continues on next page)

(continued from previous page)

```
    package_type = "python-require"
    exports = "*"
```

```
$ echo "Hello" > myfile.txt
$ conan create .
```

Now that the python-require has been created, we can access its path (the place where *myfile.txt* is) with the `path` attribute:

```
import os

from conan import ConanFile
from conan.tools.files import load

class Pkg(ConanFile):
    python_requires = "pyreq/0.1"

    def build(self):
        pyreq_path = self.python_requires["pyreq"].path
        myfile_path = os.path.join(pyreq_path, "myfile.txt")
        content = load(self, myfile_path)  # content = "Hello"
        self.output.info(content)
        # we could also copy the file, instead of reading it
```

Note that only `exports` works for this case, but not `exports_sources`.

Testing python-requires

It is possible to test with `test_package` a `python_require`, by adding a `test_package/conanfile.py`:

Listing 85: conanfile.py

```
from conan import ConanFile

def mynumber():
    return 42

class PyReq(ConanFile):
    name = "pyreq"
    version = "1.0"
    package_type = "python-require"
```

Listing 86: test_package/conanfile.py

```
from conan import ConanFile

class Tool(ConanFile):
    def test(self):
        pyreq = self.python_requires["common"].module
        mynumber = pyreq.mynumber()
        self.output.info("{}!!!".format(mynumber))
```

Note that the `test_package/conanfile.py` does not need any type of declaration of the `python_requires`, this is done automatically and implicitly. We can now create and test it with:

```
$ conan create .
...
pyreq/0.1 (test package): 42!!!
```

Effect in package_id

The `python_requires` will affect the `package_id` of the **consumer packages** using those dependencies. By default, the policy is `minor_mode`, which means:

- Changes to the **patch** version of the **revision** of a python-require will not affect the package ID. So depending on `"pyreq/1.2.3"` or `"pyreq/1.2.4"` will result in identical package ID (both will be mapped to `"pyreq/1.2.Z"` in the hash computation). Bump the patch version if you want to change your common code, but you don't want the consumers to be affected or to fire a re-build of the dependants.
- Changes to the **minor** version will produce a different package ID. So if you depend on `"pyreq/1.2.3"`, and you bump the version to `"pyreq/1.3.0"`, then, you will need to build new binaries that are using that new python-require. Bump the minor or major version if you want to make sure that packages requiring this python-require will be built using these changes in the code.

In most cases using a version-range `python_requires = "pyreq/[>=1.0 <2.0]"` is the right approach, because that means the **major** version bumps are not included because they would require changes in the consumers themselves. It is then possible to release a new major version of the `pyreq/2.0`, and have consumers gradually change their requirements to `python_requires = "pyreq/[>=2.0 <3.0]"`, fix the recipes, and move forward without breaking the whole project.

As with the regular `requires`, this default can be customized with the `core.package_id:default_python_mode` configuration.

It is also possible to customize the effect of `python_requires` per package, using the `package_id()` method:

```
from conan import ConanFile

class Pkg(ConanFile):
    python_requires ="pyreq/[>=1.0]"
    def package_id(self):
        self.info.python_requires.patch_mode()
```

Resolution of python_requires

There are few important things that should be taken into account when using `python_requires`:

- Python requires recipes are loaded by the interpreter just once, and they are common to all consumers. Do not use any global state in the `python_requires` recipes.
- Python requires are private to the consumers. They are not transitive. Different consumers can require different versions of the same `python-require`. Being private, they cannot be overriden from downstream in any way.
- `python_requires` cannot use regular `requires` or `tool_requires`.
- `python_requires` cannot be "aliased".
- `python_requires` can use native python `import` to other python files, as long as these are exported together with the recipe.
- `python_requires` can be used as editable packages too.

- `python_requires` are locked in lockfiles, to guarantee reproducibility, in the same way that other `requires` and `tool_requires` are locked.

Note: Best practices

- Even if `python-requires` can `python_requires` transitively other `python-requires` recipes, this is discouraged. Multiple level inheritance and reuse can become quite complex and difficult to manage, it is recommended to keep the hierarchy flat.
- Do not try to mix Python inheritance with `python_requires_extend` inheritance mechanisms, they are incompatible and can break.
- Do not use multiple inheritance for `python-requires`

7.6.2 Custom commands

It's possible to create your own Conan commands to solve self-needs thanks to Python and Conan public API powers altogether.

Location and naming

All the custom commands must be located in `[YOUR_CONAN_HOME]/extensions/commands/` folder. If you don't know where `[YOUR_CONAN_HOME]` is located, you can run **`conan config home`** to check it.

If _commands_ sub-directory is not created yet, you will have to create it. Those custom commands files must be Python files and start with the prefix `cmd_[your_command_name].py`. The call to the custom commands is like any other existing Conan one: **`conan your_command_name`**.

Scoping

It's possible to have another folder layer to group some commands under the same topic.

For instance:

```
| - [YOUR_CONAN_HOME]/extensions/commands/greet/
      | - cmd_hello.py
      | - cmd_bye.py
```

The call to those commands change a little bit: **`conan [topic_name]:your_command_name`**. Following the previous example:

```
$ conan greet:hello
$ conan greet:bye
```

Note: It's possible for only one folder layer, so it won't work to have something like `[YOUR_CONAN_HOME]/extensions/commands/topic1/topic2/cmd_command.py`

Decorators

conan_command(group=None, formatters=None)

Main decorator to declare a function as a new Conan command. Where the parameters are:

- `group` is the name of the group of commands declared under the same name. This grouping will appear executing the **conan -h** command.
- `formatters` is a dict-like Python object where the `key` is the formatter name and the value is the function instance where will be processed the information returned by the command one.

Listing 87: cmd_hello.py

```
import json

from conan.api.conan_api import ConanAPI
from conan.api.output import ConanOutput
from conan.cli.command import conan_command

def output_json(msg):
    return json.dumps({"greet": msg})


@conan_command(group="Custom commands", formatters={"json": output_json})
def hello(conan_api: ConanAPI, parser, *args):
    """
    Simple command to print "Hello World!" line
    """
    msg = "Hello World!"
    ConanOutput().info(msg)
    return msg
```

Important: The function decorated by `@conan_command(....)` must have the same name as the suffix used by the Python file. For instance, the previous example, the file name is `cmd_hello.py`, and the command function decorated is `def hello(....)`.

conan_subcommand(formatters=None)

Similar to `conan_command`, but this one is declaring a sub-command of an existing custom command. For instance:

Listing 88: cmd_hello.py

```
from conan.api.conan_api import ConanAPI
from conan.api.output import ConanOutput
from conan.cli.command import conan_command, conan_subcommand


@conan_subcommand()
def hello_moon(conan_api, parser, subparser, *args):
    """
    Sub-command of "hello" that prints "Hello Moon!" line
    """
    ConanOutput().info("Hello Moon!")
```

(continues on next page)

(continued from previous page)

```
@conan_command(group="Custom commands")
def hello(conan_api: ConanAPI, parser, *args):
    """
    Simple command "hello"
    """
```

The command call looks like **conan hello moon**.

Note: Notice that to declare a sub-command is required an empty Python function acts as the main command.

Formatters arguments

The return of the command will be passed as argument to the formatters. If there are different formatters that require different arguments, the approach is to return a dictionary, and let the formatters chose the arguments they need. For example, the `graph info` command uses several formatters like:

```
def format_graph_html(result):
    graph = result["graph"]
    conan_api = result["conan_api"]
    ...

def format_graph_info(result):
    graph = result["graph"]
    field_filter = result["field_filter"]
    package_filter = result["package_filter"]
    ...

@conan_subcommand(formatters={"text": format_graph_info,
                              "html": format_graph_html,
                              "json": format_graph_json,
                              "dot": format_graph_dot})
def graph_info(conan_api, parser, subparser, *args):
    ...
    return {"graph": deps_graph,
            "field_filter": args.filter,
            "package_filter": args.package_filter,
            "conan_api": conan_api}
```

Command function arguments

These are the passed arguments to any custom command and its sub-commands functions:

Listing 89: cmd_command.py

```
from conan.cli.command import conan_command, conan_subcommand


@conan_subcommand()
def command_subcommand(conan_api, parser, subparser, *args):
    """
    subcommand information. This info will appear on ``conan command subcommand -h``.
```

(continues on next page)

(continued from previous page)

```
    :param conan_api: <object conan.api.conan_api.ConanAPI> instance
    :param parser: root <object argparse.ArgumentParser> instance (coming from main␣
↪command)
    :param subparser: <object argparse.ArgumentParser> instance for sub-command
    :param args: ``list`` of all the arguments passed after sub-command call
    :return: (optional) whatever is returned will be passed to formatters functions␣
↪(if declared)
    """
    # ...


@conan_command(group="Custom commands")
def command(conan_api, parser, *args):
    """
    command information. This info will appear on ``conan command -h``.

    :param conan_api: <object conan.api.conan_api.ConanAPI> instance
    :param parser: root <object argparse.ArgumentParser> instance
    :param args: ``list`` of all the arguments passed after command call
    :return: (optional) whatever is returned will be passed to formatters functions␣
↪(if declared)
    """
    # ...
```

- `conan_api`: instance of `ConanAPI` class. See more about it in *conan.api.conan_api.ConanAPI section*
- `parser`: root instance of Python `argparse.ArgumentParser` class to be used by the main command function. See more information in argparse official website.
- `subparser` (only for sub-commands): child instance of Python `argparse.ArgumentParser` class for each sub-command function.
- `*args`: list of all the arguments passed via command line to be parsed and used inside the command function. Normally, they'll be parsed as `args = parser.parse_args(*args)`. For instance, running **conan mycommand arg1 arg2 arg3**, the command function will receive them as a Python list-like `["arg1", "arg2", "arg3"]`.

Read more

- *Custom command to remove recipe and package revisions but the latest package one from the latest recipe one*.

7.6.3 Python API

Warning: The full Python API is **experimental**. See *the Conan stability* section for more information.

Conan API Reference

Warning: This feature is experimental and subject to breaking changes.

class ConanAPI(*cache_folder=None*)

Read more

- Creating Conan custom commands
- ...

Remotes API

> **Warning:** This feature is experimental and subject to breaking changes.

class RemotesAPI(*conan_api*)

list(*pattern=None*, *only_enabled=True*)

Parameters

- **pattern** – if None, all remotes will be listed it can be a single value or a list of values
- **only_enabled** –

Returns

Search API

> **Warning:** This feature is experimental and subject to breaking changes.

class SearchAPI(*conan_api*)

List API

> **Warning:** This feature is experimental and subject to breaking changes.

class ListAPI(*conan_api*)

Get references from the recipes and packages in the cache or a remote

static filter_packages_configurations(*pkg_configurations*, *query*)

Parameters

- **pkg_configurations** – Dict[PkgReference, PkgConfiguration]
- **query** – str like "os=Windows AND (arch=x86 OR compiler=gcc)"

Returns Dict[PkgReference, PkgConfiguration]

Profiles API

Warning: This feature is experimental and subject to breaking changes.

class ProfilesAPI(*conan_api*)

get_default_host()

Returns the path to the default "host" profile, either in the cache or as defined by the user in configuration

get_default_build()

Returns the path to the default "build" profile, either in the cache or as defined by the user in configuration

get_profile(*profiles*, *settings=None*, *options=None*, *conf=None*, *cwd=None*)

Computes a Profile as the result of aggregating all the user arguments, first it loads the "profiles", composing them in order (last profile has priority), and finally adding the individual settings, options (priority over the profiles)

get_path(*profile*, *cwd=None*, *exists=True*)

Returns the resolved path of the given profile name, that could be in the cache, or local, depending on the "cwd"

list()

List all the profiles file sin the cache :return: an alphabetically ordered list of profile files in the default cache location

static detect()

Returns an automatically detected Profile, with a "best guess" of the system settings

Install API

Warning: This feature is experimental and subject to breaking changes.

class InstallAPI(*conan_api*)

install_binaries(*deps_graph*, *remotes=None*)

Install binaries for dependency graph :param deps_graph: Dependency graph to intall packages for :param remotes:

install_system_requires(*graph*, *only_info=False*)

Install binaries for dependency graph :param only_info: Only allow reporting and checking, but never install :param graph: Dependency graph to intall packages for

install_sources(*graph*, *remotes*)

Install sources for dependency graph :param remotes: :param graph: Dependency graph to install packages for

install_consumer(*deps_graph*, *generators=None*, *source_folder=None*, *output_folder=None*, *deploy=False*)

Once a dependency graph has been installed, there are things to be done, like invoking generators for the root consumer. This is necessary for example for conanfile.txt/py, or for "conan install <ref> -g

Graph API

> **Warning:** This feature is experimental and subject to breaking changes.

class GraphAPI(*conan_api*)

load_root_test_conanfile(*path*, *tested_reference*, *profile_host*, *profile_build*, *update=None*, *remotes=None*, *lockfile=None*, *tested_python_requires=None*)

Create and initialize a root node from a test_package/conanfile.py consumer

Parameters

- **tested_python_requires** – the reference of the `python_require` to be tested
- **lockfile** – Might be good to lock python-requires, build-requires
- **path** – The full path to the test_package/conanfile.py being used
- **tested_reference** – The full RecipeReference of the tested package
- **profile_host** –
- **profile_build** –
- **update** –
- **remotes** –

Returns a graph Node, recipe=RECIPE_CONSUMER

load_graph(*root_node*, *profile_host*, *profile_build*, *lockfile=None*, *remotes=None*, *update=False*, *check_update=False*)

Compute the dependency graph, starting from a root package, evaluation the graph with the provided configuration in profile_build, and profile_host. The resulting graph is a graph of recipes, but packages are not computed yet (package_ids) will be empty in the result. The result might have errors, like version or configuration conflicts, but it is still possible to inspect it. Only trying to install such graph will fail

Parameters

- **root_node** – the starting point, an already initialized Node structure, as returned by the "load_root_node" api
- **profile_host** – The host profile
- **profile_build** – The build profile
- **lockfile** – A valid lockfile (None by default, means no locked)
- **remotes** – list of remotes we want to check
- **update** – (False by default), if Conan should look for newer versions or revisions for already existing recipes in the Conan cache
- **check_update** – For "graph info" command, check if there are recipe updates

analyze_binaries (*graph*, *build_mode=None*, *remotes=None*, *update=None*, *lockfile=None*)
Given a dependency graph, will compute the package_ids of all recipes in the graph, and evaluate if they should be built from sources, downloaded from a remote server, of if the packages are already in the local Conan cache

Parameters

- **lockfile** –
- **graph** – a Conan dependency graph, as returned by "load_graph()"
- **build_mode** – TODO: Discuss if this should be a BuildMode object or list of arguments
- **remotes** – list of remotes
- **update** – (False by default), if Conan should look for newer versions or revisions for already existing recipes in the Conan cache

load_conanfile_class (*path*)
Given a path to a conanfile.py file, it loads its class (not instance) to allow inspecting the class attributes, like 'name', 'version', 'description', 'options' etc

Export API

> **Warning:** This feature is experimental and subject to breaking changes.

class ExportAPI (*conan_api*)

Remove API

> **Warning:** This feature is experimental and subject to breaking changes.

class RemoveAPI (*conan_api*)

Config API

> **Warning:** This feature is experimental and subject to breaking changes.

class ConfigAPI (*conan_api*)

New API

> **Warning:** This feature is experimental and subject to breaking changes.

class NewAPI (*conan_api*)

get_template(*template_folder*)
Load a template from a user absolute folder

get_home_template(*template_name*)
Load a template from the Conan home templates/command/new folder

Upload API

> **Warning:** This feature is experimental and subject to breaking changes.

class UploadAPI(*conan_api*)

check_upstream(*package_list*, *remote*, *force=False*)
Check if the artifacts are already in the specified remote, skipping them from the package_list in that case

prepare(*package_list*, *enabled_remotes*)
Compress the recipes and packages and fill the upload_data objects with the complete information. It doesn't perform the upload nor checks upstream to see if the recipe is still there

Download API

> **Warning:** This feature is experimental and subject to breaking changes.

class DownloadAPI(*conan_api*)

7.6.4 Deployers

Deployers are a mechanism to facilitate copying files form one folder, usually the Conan cache, to user folders. While Conan provides two built-in ones (`full_deploy` and `direct_deploy`), users can easily manage their own with `conan config install`.

Deployers run before generators, and they can change the target folders. For example, if the `--deploy=full_deploy` deployer runs before `CMakeDeps`, the files generated by `CMakeDeps` will point to the local copy in the user folder done by the `full_deploy` deployer, and not to the Conan cache. Multiple deployers can be specified by supplying more than one `--deploy=` argument, and they will be ran in order of appearance.

Deployers can be multi-configuration. Running `conan install . --deploy=full_deploy` repeatedly for different profiles can achieve a fully self-contained project, including all the artifacts, binaries, and build files. This project will be completely independent of Conan and no longer require it at all to build.

Built-in deployers

full_deploy

Deploys each package folder of every dependency to your recipe's `output_folder` in a subfolder tree based on:

1. The build context
2. The dependency name and version

3. The build type
4. The build arch

Then every dependency will end up in a folder such as:

[OUTPUT_FOLDER]/host/dep/0.1/Release/x86_64

direct_deploy

Same as `full_deploy`, but only processes your recipe's *direct* dependencies.

> **Warning:** The built-in deployers are in **preview**. See *the Conan stability* section for more information.

Custom deployers

Custom deployers can be managed via `conan config install`. When looking for a specific deployer, Conan will look in these locations for the deployer in the following order:

1. Absolute paths
2. Relative to cwd
3. In the `[CONAN_HOME]/extensions/deploy` folder
4. As built-in deployers

Conan will look for a `deploy()` method to call for each installed file. The function signature of your custom deployers should be as follows:

Listing 90: **my_custom_deployer.py**

```
def deploy(graph, output_folder: str):
```

(Note that the arguments are passed as named parameters, so both the `graph` and `output_folder` names are mandatory)

You can access your conanfile object with `graph.root.conanfile`. See *ConanFile.dependencies* for information on how to iterate over its dependencies. Your custom deployer can now be invoked as if it were a built-in deployer using the filename in which it's found, in this case `conan install . --deploy=my_custom_deployer`. Note that supplying the **.py** extension is optional.

See the *custom deployers* section for examples on how to implement your own deployers.

7.6.5 Hooks

The Conan hooks is a feature intended to extend the Conan functionalities to perform certain orthogonal operations, like some quality checks, in different stages of a package creation process, like pre-build and post-build.

Hook structure

A hook is a Python function that will be executed at certain points of Conan workflow to customize the client behavior without modifying the client sources or the recipe ones.

Here is an example of a simple hook:

Listing 91: *hook_example.py*

```python
from conan.tools.files import load

def pre_export(conanfile):
    for field in ["url", "license", "description"]:
        field_value = getattr(conanfile, field, None)
        if not field_value:
            conanfile.output.error(f"[REQUIRED ATTRIBUTES] Conanfile doesn't have '
↪{field}'.
                                    It is recommended to add it as attribute.")
```

This hook checks the recipe content prior to it being exported. Basically the `pre_export()` function checks the attributes of the `conanfile` object to see if there is an URL, a license and a description and if missing, warns the user with a message through the `conanfile.output`. This is done **before** the recipe is exported to the local cache.

Any kind of Python script can be executed. You can create global functions and call them from different hook functions, import from a relative module and warn, error or even raise to abort the Conan client execution.

Importing from a module

The hook interface should always be placed inside a Python file with the name of the hook starting by *hook_* and with the extension *.py*. It also should be stored in the *<conan_home>/extensions/hooks* folder. However, you can use functionalities from imported modules if you have them installed in your system or if they are installed with Conan:

Listing 92: hook_example.py

```python
import requests
from conan.tools.files import replace_in_file

def post_package(conanfile):
    if not os.path.isdir(os.path.join(conanfile.package_folder, "licenses")):
        response = requests.get('https://api.github.com/repos/company/repository/
↪contents/LICENSE')
```

You can also import functionalities from a relative module:

```
hooks
├── custom_module
│   ├── custom.py
│   └── __init__.py
└── hook_printer.py
```

Inside the *custom.py* from my *custom_module* there is:

Listing 93: custom.py

```python
def my_printer(conanfile):
    conanfile.output.info("my_printer(): CUSTOM MODULE")
```

And it can be used in the hook importing the module, just like regular Python:

Listing 94: hook_printer.py

```python
from custom_module.custom import my_printer

```

(continues on next page)

(continued from previous page)

```
def pre_export(conanfile):
    my_printer(conanfile)
```

Hook interface

Here you can see a complete example of all the hook functions available:

Listing 95: *hook_full.py*

```
def pre_export(conanfile):
    conanfile.output.info("Running before to execute export() method.")

def post_export(conanfile):
    conanfile.output.info("Running after of executing export() method.")

def pre_source(conanfile):
    conanfile.output.info("Running before to execute source() method.")

def post_source(conanfile):
    conanfile.output.info("Running after of executing source() method.")

def pre_generate(conanfile):
    conanfile.output.info("Running before to execute generate() method.")

def post_generate(conanfile):
    conanfile.output.info("Running after of executing generate() method.")

def pre_build(conanfile):
    conanfile.output.info("Running before to execute build() method.")

def post_build(conanfile):
    conanfile.output.info("Running after of executing build() method.")

def pre_package(conanfile):
    conanfile.output.info("Running before to execute package() method.")

def post_package(conanfile):
    conanfile.output.info("Running after of executing package() method.")

def pre_package_info(conanfile):
    conanfile.output.info("Running before to execute package_info() method.")

def post_package_info(conanfile):
    conanfile.output.info("Running after of executing package_info() method.")
```

Functions of the hooks are intended to be self-descriptive regarding to the execution of them. For example, the `pre_package()` function is called just before the `package()` method of the recipe is executed.

All hook methods are filled only with the same single object:

- **conanfile**: It is a regular `ConanFile` object loaded from the recipe that received the Conan command. It has its normal attributes and dynamic objects such as `build_folder`, `package_folder`, `output`, `dependencies`, `options` ...

Storage, activation and sharing

Hooks are Python files stored under *<conan_home>/extensions/hooks* folder and **their file name should start with hook_ and end with the .py extension**.

The activation of the hooks is done automatically once the hook file is stored in the hook folder. In case storing in subfolders, it works automatically too.

To deactivate a hook, its file should be removed from the hook folder. There is no configuration which can deactivate but keep the file stored in hooks folder.

Official Hooks

There are some officially maintained hooks in its own repository in Conan hooks GitHub, but mostly are only compatible with Conan 1.x, so please, check first the README to have information which hooks are compatible with Conan v2.

7.6.6 Binary compatibility

This plugin, located in the cache `extensions/plugins/compatibility/compatibility.py` allows defining custom rules for the binary compatibility of packages accross settings and options. It has some built-in logic implemented, but can be customized.

```
def compatibility(conanfile):
    result = []
    if conanfile.settings.build_type == "Debug":
        result.append({"settings": [("build_type", "Release")]})
    return result
```

Some important rules:

- The built-in `compatibility.py` is subject to changes in future releases. To avoid being updated in the future, please remove the first comment `# This file was generated by Conan.`

Warning: The `compatibility.py` feature is in **preview**. The current default `compatibility.py` is **experimental**. See *the Conan stability* section for more information.

7.6.7 Profile plugin

The `profile.py` extension plugin is a Python script that receives one profile and allow checking and modifying it.

This plugin is located in the `extensions/plugins/profile.py` cache folder.

This `profile.py` contains a default implementation that does:

- Will try to define `compiler.runtime_type` for `msvc` compiler if it is not defined, and it will define it to match the `settings.build_type`. That allow users to let it undefined in profiles, and switch it conveniently in command line just with `-s build_type=Debug`
- Will check the `compiler.cppstd` value if defined to validate if the current compiler version has support for it. For example, if a developer tries to use `-s compiler=gcc -s compiler.version=5 -s compiler.cppstd=20`, it will raise an error.

Users can customize this `profile.py` and distribute it via `conan config install`, in that case, the first lines should be removed:

```
# This file was generated by Conan. Remove this comment if you edit this file or Conan
# will destroy your changes.
```

And `profile.py` should contain one function with the signature:

```
def profile_plugin(profile):
    settings = profile.settings
    print(settings)
```

When a profile is computed, it will display something like:

```
OrderedDict([('arch', 'x86_64'), ('build_type', 'Release'), ('compiler', 'msvc'), (
↪'compiler.cppstd', '14'), ('compiler.runtime', 'dynamic'), ('compiler.runtime_type',
↪ 'Release'), ('compiler.version', '192'), ('os', 'Windows')])
```

See also:

- See the documentation about the *Conan profiles*.

7.6.8 Command wrapper

The `cmd_wrapper.py` extension plugin is a Python script that receives the command line argument provided by `self.run()` recipe calls, and allows intercepting them and returning a new one.

This plugin must be located in the `extensions/plugins` cache folder, and can be installed with the `conan config install` command.

For example:

```
def cmd_wrapper(cmd):
    return 'echo "{}"'.format(cmd)
```

Would just intercept the commands and display them to terminal, which means that all commmands in all recipes `self.run()` will not execute, but just be echoed.

A more common use case would be the injection of a parallelization tools over some commands, which could look like:

```
def cmd_wrapper(cmd):
    # lets parallellize only CMake invocations
    if cmd.startswith("cmake"):
        return 'parallel-build "{}"  --parallel-argument'.format(cmd)
    # otherwise return same command, not modified
    return cmd
```

7.6.9 Package signing

> **Warning:** The package signing plugin is in **preview**. See *the Conan stability* section for more information.

This plugin, which must be located in the cache `extensions/plugins/sign/sign.py` file contains 2 methods:

- The `sign(ref, artifacts_folder, signature_folder)` executes for every recipe and package that is to be uploaded to a server. The `ref` is the full reference to the artifact, it can be either a recipe reference or a package reference. The `artifacts_folder` is the folder containing the files to be uploaded, typically the `conanfile.py`, `conan_package.tgz`, `conanmanifest.txt`, etc. The `signature_folder` contains the folder in which the generated files should be written.

- The `verify(ref, artifacts_folder, signature_folder)` executes when a package is installed from a server, receives the same arguments as above and should be used to verify the integrity or correctness of the signatures

Example of a package signer that puts the artifact filenames in a file called `signature.asc` when the package is uploaded and assert that the downloaded artifacts are in the downloaded `signature.asc`:

```
import os

def sign(ref, artifacts_folder, signature_folder):
    print("Signing ref: ", ref)
    print("Signing folder: ", artifacts_folder)
    files = []
    for f in sorted(os.listdir(artifacts_folder)):
        if os.path.isfile(os.path.join(artifacts_folder, f)):
            files.append(f)
    signature = os.path.join(signature_folder, "signature.asc")
    open(signature, "w").write("\n".join(files))

def verify(ref, artifacts_folder, signature_folder):
    print("Verifying ref: ", ref)
    print("Verifying folder: ", artifacts_folder)
    signature = os.path.join(signature_folder, "signature.asc")
    contents = open(signature).read()
    print("verifying contents", contents)
    for f in sorted(os.listdir(artifacts_folder)):
        print("VERIFYING ", f)
        if os.path.isfile(os.path.join(artifacts_folder, f)):
            assert f in contents
```

7.7 Environment variables

These are very few environment variables that can be used to configure some of the Conan behavior. These variables are the exception, for customization and configuration control, Conan uses the *global.conf configuration* and the *profile [conf] section*

7.7.1 CONAN_HOME

This variable controls the location of the Conan home folder. By default, if it is not defined, it will be `<username>/.conan2`.

Note: Recall that the Conan package cache, contained in the Conan home, is not concurrent. Different parallel tasks like those that can happen in CI, need to use a separate cache, and defining `CONAN_HOME` is the way to do it.

7.7.2 CONAN_DEFAULT_PROFILE

The default profile will be the `"default"` file in the Conan cache. This environment variable allows to define a different default name. There are also `conf` items `core:default_profile` and `core:default_build_profile` to define such default profile names, this env-var should be used only when the `conf` is not enough.

7.7.3 Remote login variables

`CONAN_LOGIN_USERNAME`, `CONAN_LOGIN_USERNAME_{REMOTE_NAME}` define the login username for a given remote. `CONAN_PASSWORD`, `CONAN_PASSWORD_{REMOTE_NAME}` define the login password for a given remote.

These environment variables are just a substitute of the interactive input of the username or password when Conan CLI requests it. They do not perform any kind of authentication unless the remote server throws an authentication challenge. That means that for some remote servers configured to allow anonymous usage, these will not be used, and the user will remain as an unauthenticated user, unless a `conan remote login` or `conan remote auth` is done first.

When the Conan CLI is about to ask the user for the remote password, it will check the variable `CONAN_LOGIN_USERNAME_{REMOTE_NAME}` or `CONAN_PASSWORD_{REMOTE_NAME}` first, if the variable is not declared Conan will try to use the variable `CONAN_LOGIN_USERNAME` and `CONAN_PASSWORD` respectively, if the variable is not declared either, Conan will request to the user to input a password or fail.

The remote name is transformed to all uppercase. If the remote name contains "-", you have to replace it with "_" in the variable name.

Note:

- These variables are useful for unattended executions like CI servers or automated tasks, as CI secrets
- These variables are not recommended for developer machines.
- Recall that these variables do not perform authentication unless the remote server requests it.
- The `core:non_interactive` conf can be defined in `global.conf` to force Conan to fail if any interactive prompt is requested, to avoid CI process being stuck.

7.7.4 Terminal color variables

Conan default behavior is try to autodetect the output. If the output is redirected to a file, or other support not `tty`, that cannot print colors, it will disable colored output. For regular terminals, it will try to do colored output, unless some of the following change that behavior:

- `CLICOLOR_FORCE` Forces the generation of terminal color escape characters, no matter what the autodetection of terminal is.
- `NO_COLOR` disables the generation of color escape characters. This will be ignored if `CLICOLOR_FORCE` is activated.
- `CONAN_COLOR_DARK` will revert the color scheme for white/light background terminals (default assumes dark background).

7.8 Conan Server

Important: This server is mainly used for testing (though it might work fine for small teams). We recommend using the free *Artifactory Community Edition for C/C++* for private development or **Artifactory Pro** as Enterprise solution.

7.8.1 Configuration

By default your server configuration is saved under `~/.conan_server/server.conf`, however you can modify this behaviour by either setting the `CONAN_SERVER_HOME` environment variable or launching the server with `-d` or `--server_dir` command line argument followed by desired path. In case you use one of the options your configuration file will be stored under `server_directory/server.conf` Please note that command line argument will override the environment variable. You can change configuration values in `server.conf`, prior to launching the server. Note that the server does not support hot-reload, and thus in order to see configuration changes you will have to manually relaunch the server.

The server configuration file is by default:

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

ssl_enabled: False
port: 9300

public_port:
host_name: localhost

authorize_timeout: 1800

disk_storage_path: ./data
disk_authorize_timeout: 1800
updown_secret: HJhjujkjkjkJKLUYyuuyHJ

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

[read_permissions]
*/*@*/*: *

[users]
demo: demo
```

Server Parameters

Note: The Conan server supports relative URLs, allowing you to 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. This allows accessing 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.

- 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 into account if your server has multiple network interfaces.
- public_port: Might be needed when running virtualized, Docker or any other kind of port redirection. File 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, the port value is used.

  Example: Use conan_server in a Docker container that internally runs in the 9300 port but 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 the conan_server will generate the upload/download urls with "https" instead of "http".

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

1. Use the **conan remote** command to disable the SSL certificate checks. E.g., *conan remote add/update myremote https://somedir False*
2. If using the *core.net.http:cacert_path* configuration in the Conan client, append the server *.crt* file contents to the *cacert.pem* location.

The folder in which the uploaded packages are stored (i.e., the folder you would want to backup) is defined in the 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 sufficient so that 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.

Permissions Parameters

By default, the server configuration when set to Read can be done anonymous, but uploading requires you to be registered users. Users can easily be registered in the [users] section, by defining a pair of login: password for each one. Plain text passwords are used 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 the [read_permissions] and [write_permissions] sections. These sections specify the sequence of patterns and authorized 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 continue searching for matches.

Authentication

By default, Conan provides 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 two-step process:

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

This is a list of available authenticators, visit their URLs to retrieve 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:

```python
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.

Authorizations

By default, Conan uses the contents of the `[read_permissions]` and `[write_permissions]` sections to authorize or reject a request.

A plugin system is also available to customize the authorization mechanism. The installation of such a plugin is a simple two-step process:

1. Copy the authorizer's source file into the `.conan_server/plugins/authorizer` folder.
2. Add `custom_authorizer: authorizer_name` to the `server.conf` [server] section.

Create Your Own Custom Authorizer

If you want to create your own Authorizer, create a Python module in `~/.conan_server/plugins/authorizer/my_authorizer.py`

Example:

```
from conans.errors import AuthenticationException, ForbiddenException

def get_class():
    return MyAuthorizer()

class MyAuthorizer(object):
    def _check_conan(self, username, ref):
        if ref.user == username:
            return

        if username:
            raise ForbiddenException("Permission denied")
        else:
            raise AuthenticationException()

    def _check_package(self, username, pref):
       self._check(username, pref.ref)

    check_read_conan = _check_conan check_write_conan = _check_conan
    check_delete_conan = _check_conan check_read_package = _check_package
    check_write_package = _check_package check_delete_package = _check_package
```

The module has to implement:

- A factory function `get_class()` that returns an instance of a class conforming to the Authorizer's interface.
- **A class that implements all the methods defined in the Authorizer interface:**
  - `check_read_conan()` is used to decide whether to allow read access to a recipe.
  - `check_write_conan()` is used to decide whether to allow write access to a recipe.
  - `check_delete_conan()` is used to decide whether to allow a recipe's deletion.
  - `check_read_package()` is used to decide whether to allow read access to a package.
  - `check_write_package()` is used to decide whether to allow write access to a package.
  - `check_delete_package()` is used to decide whether to allow a package's deletion.

The `check_*_conan()` methods are called with a username and `conans.model.ref.ConanFileReference` instance as their arguments. Meanwhile the `check_*_package()` methods are passed a username and `conans.model.ref.PackageReference` instance as their arguments. These methods should raise an exception, unless the user is allowed to perform the requested action.

7.8.2 Running the 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
```

7.8.3 Running the 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;

       location /subdir/ {
          proxy_pass http://0.0.0.0:9300/;
       }
  }
```

remote configuration in Conan client

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

7.8.4 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 similar to the following example:

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

```
<VirtualHost *:80>
    WSGIScriptAlias /
    /usr/local/lib/python3.6/dist-packages/conans/server/server_launcher.py
    WSGICallableObject app WSGIPassAuthorization On

    <Directory /usr/local/lib/python3.6/dist-packages/conans>
```

(continues on next page)

(continued from previous page)

```
        Require all granted
    </Directory>
</VirtualHost>
```

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

Apache conf file (e.g., /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.g., var/www/.conan_server, so remember to use that directory to configure your Conan server.

See also:

- *Setting-up a Conan Server*

CHAPTER

EIGHT

KNOWLEDGE

8.1 Core guidelines

8.1.1 Good practices

- **build() should be simple, prepare the builds in generate() instead**: The recipes' `generate()` method purpose is to prepare the build as much as possible. Users calling `conan install` will execute this method, and the generated files should allow users to do "native" builds (calling directly "cmake", "meson", etc.) as easy as possible. Thus, avoiding as much as possible any logic in the `build()` method, and moving it to the `generate()` method helps developers achieve the same build locally as the one that would be produced by a `conan create` build in the local cache.
- **Always use your own profiles in production**, instead of relying on the auto-detected profile, as the output of such auto detection can vary over time, resulting in unexpected results. Profiles (and many other configuration), can be managed with `conan config install`.
- **Developers should not be able to upload to "development" and "production" repositories** in the server. Only CI builds have write permissions in the server. Developers should only have read permissions and at most to some "playground" repositories used to work and share things with colleagues, but which packages are never used, moved or copied to the development or production repositories.
- The `test_package` purpose is to check that the package has been correctly created (that is, that it has correctly packaged the headers, the libraries, etc, in the right folders), not that the functionality of the package is correct. Then, it should be kept as simple as possible, like building and running an executable that uses the headers and links against a packaged library should be enough. Such execution should be as simple as possible too. Any kind of unit and functional tests should be done in the `build()` method.
- **Keep ''python_requires'' as simple as possible**. Avoid transitive `python_requires`, keep them as reduced as possible, and at most, require them explicitly in a "flat" structure, without `python_requires` requiring other `python_requires`. Avoid inheritance (via `python_requires_extend`) if not strictly necessary, and avoid multiple inheritance at all costs, as it is extremely complicated, and it does not work the same as the built-in Python one.
- At the moment the **Conan cache is not concurrent**. Avoid any kind of concurrency or parallelism, for example different parallel CI jobs should use different caches (with CONAN_HOME env-var). This might change in the future and we will work on providing concurrency in the cache, but until then, use isolated caches for concurrent tasks.

8.1.2 Forbidden practices

- **Conan is not re-entrant**: Calling the Conan process from Conan itself cannot be done. That includes calling Conan from recipe code, hooks, plugins, and basically every code that already executes when Conan is called. Doing it will result in undefined behavior. For example it is not valid to run `conan search` from a `conanfile.py`. This includes indirect calls, like running Conan from a build script (like `CMakeLists.txt`) while this build script is already being executed as a result of a Conan invocation. For the same reason **Conan Python API cannot be used from recipes**: The Conan Python API can only be called from Conan custom commands or from user Python scripts, but never from `conanfile.py` recipes, hooks, extensions, plugins, or any other code executed by Conan.

- **Recipes reserved names**: Conan `conanfile.py` recipes user attributes and methods should always start with _. Conan reserves the "public" namespace for all attributes and methods, and `_conan` for private ones. Using any non-documented Python function, method, class, attribute, even if it is "public" in the Python sense, is undefined behavior if such element is not documented in this documentation.
- **Conan artifacts are immutable**: Conan packages and artifacts, once they are in the Conan cache, they are assumed to be immutable. Any attempt to modify the exported sources, the recipe, the conandata.yml or any of the exported or the packaged artifacts, is undefined behavior. For example, it is not possible to modify the contents of a package inside the `package_info()` method or the `package_id()` method, those methods should never modify, delete or create new files inside the packages. If you need to modify some package, you might use your own custom `deployer`.
- **Conan cache paths are internal implementation detail**: The Conan cache paths are an internal implementation detail. Conan recipes provide abstractions like `self.build_folder` to represent the necessary information about folders, and commands like `conan cache path` to get information of the current folders. The Conan cache might be checked while debugging, as read-only, but it is not allowed to edit, modify or delete artifacts or files from the Conan cache by any other means that Conan command line or public API.

8.2 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!

8.2.1 Troubleshooting

ERROR: Missing prebuilt package

When installing packages (with **`conan install`** or **`conan create`**) it is possible that you get an error like the following one:

```
ERROR: Missing binary: zlib/1.2.11:b1d267f77ddd5d10d06d2ecf5a6bc433fbb7eeed

zlib/1.2.11: WARN: Can't find a 'zlib/1.2.11' package binary
↪'b1d267f77ddd5d10d06d2ecf5a6bc433fbb7eeed' for the configuration:
[settings]
arch=x86_64
build_type=Release
compiler=apple-clang
compiler.cppstd=gnu11
compiler.libcxx=libc++
compiler.version=14
os=Macos
[options]
fPIC=True
shared=False
```

(continues on next page)

(continued from previous page)

```
ERROR: Missing prebuilt package for 'zlib/1.2.11'
Check the available packages using 'conan list zlib/1.2.11:* -r=remote'
or try to build locally from sources using the '--build=zlib/1.2.11' argument

More Info at 'https://docs.conan.io/en/2/knowledge/faq.html#error-missing-prebuilt-
↪package'
```

This means that the package recipe `zlib/1.2.11` exists, but for some reason there is no precompiled package for your current settings or options. Maybe the package creator didn't build and shared pre-built packages at all and only uploaded the package recipe, or they are only providing packages for some platforms or compilers. E.g. the package creator built packages from the recipe for apple-clang 11, but you are using apple-clang 14. Also you may want to check your *package ID mode* as it may have an influence on the packages available for it.

By default, Conan doesn't build packages from sources. There are several possibilities to overcome this error:

- You can try to build the package for your settings from sources, indicating some build policy as argument, like **`--build zlib*`** 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 fails, and you are using the *conancenter* remote, you can open an issue in the Conan Center Index repository

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']
```

This doesn't mean that such compiler version 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 `~/.conan2/settings.yml` a settings file that you can modify, edit, add any setting or any value, with any nesting if necessary. See *settings.yml* to learn how you can customize your settings to model your binaries at your will.

As long as your team or users have the same settings (`settings.yml` and `settings_user.yml` an be easily shared with the `conan config install` command), everything will work. The *settings.yml* file is just a mechanism so users agree on a common spelling for typical 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 built-in helper or integrations, like `CMake` or `CMakeToolchain` will not understand the new added settings, don't use them or even fail if you added some new unexpected value to existing settings. 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.

8.3 Videos

> **Warning:** This section presents some conference talks and presentations regarding Conan. While they can be very informative and educational, please note that some of them might be outdated. Always use the documentation and reference as the source of truth, not the videos.

- ACCU 2022: Advanced Dependencies Model in Conan 2.0 C, C++ Package Manager by Diego Rodriguez-Losada

CHAPTER

NINE

CHANGELOG

For a more detailed description of the major changes that Conan 2.0 brings, compared with Conan 1.X, please read *What's new in Conan 2.0*

9.1 2.0.1 (03-Mar-2023)

- Feature: Add *–insecure* alias to *–verify-ssl* in config install. #13270 . Docs here
- Feature: Add *.conanignore* support to *conan config install*. #13269 . Docs here
- Feature: Make verbose tracebacks on exception be shown for `-vv` and `-vvv`, instead of custom env-var used in 1.X. #13226
- Fix: Minor improvements to **`conan install`** and 2.0-readiness error messages. #13299
- Fix: Remove `vcvars.bat` VS telemetry env-var, to avoid Conan hanging. #13293
- Fix: Remove legacy `CMakeToolchain` support for `CMakePresets` schema2 for `CMakeUserPresets.json`. #13288 . Docs here
- Fix: Remove `--logger` json logging and legacy traces. #13287 . Docs here
- Fix: Fix typo in *conan remote auth* help. #13285 . Docs here
- Fix: Raise arg error if `conan config list unexpected-arg`. #13282
- Fix: Do not auto-detect `compiler.runtime_type` for `msvc`, rely on profile plugin. #13277
- Fix: Fix conanfile.txt options parsing error message. #13266
- Fix: Improve error message for unified patterns in options. #13264
- Fix: Allow `conan remote add --force` to force re-definition of an existing remote name. #13249
- Fix: Restore printing of profiles for build command. #13214
- Fix: Change **`conan build`** argument description for "path" to indicate it is only for conanfile.py and explicitly state that it does not work with conanfile.txt. #13211 . Docs here
- Fix: Better error message when dependencies `options` are defined in `requirements()` method. #13207
- Fix: Fix broken links to docs from error messages and readme. #13186
- Bugfix: Ensure that *topics* are always serialized as lists. #13298
- Bugfix: Ensure that *provides* are always serialized as lists. #13298
- Bugfix: Fixed the detection of certain visual c++ installations. #13284
- Bugfix: Fix supported `cppstd` values for `msvc` compiler. #13278

- Bugfix: CMakeDeps generate files for `tool_requires` with the same `build_type` as the "host" context. #13267
- Bugfix: Fix definition of patterns for dependencies options in configure(). #13263
- Bugfix: Fix CMakeToolchain error when output folder in different Win drive. #13248
- Bugfix: Do not raise errors if a `test_requires` is not used by components `.requires`. #13191

9.2 2.0.0 (22-Feb-2023)

- Feature: Change default profile cppstd for apple-clang to gnu17. #13185
- Feature: New `conan remote auth` command to force authentication in the remotes #13180
- Fix: Allow defining options trait in `test_requires(..., options={})` #13178
- Fix: Unifying Conan commands help messages. #13176
- Bugfix: Fix MesonToolchain wrong cppstd in apple-clang #13172
- Feature: Improved global Conan output messages (create, install, export, etc.) #12746

9.3 2.0.0-beta10 (16-Feb-2023)

- Feature: Add basic html output to *conan list* command. #13135
- Feature: Allow `test_package` to process `--build` arguments (computing –build=never for the main, non test_package graph). #13117
- Feature: Add *–force* argument to remote add. #13112
- Feature: Validate if the input configurations exist, to avoid typos. #13110
- Feature: Allow defining `self.folders.build_folder_vars` in recipes `layout()`. #13109
- Feature: Block settings assignment. #13099
- Feature: Improve *conan editable* ui. #13093
- Feature: Provide the ability for users to extend Conan generated CMakePresets. #13090
- Feature: Add error messages to help with the migration of recipes to 2.0, both from ConanCenter and from user repos. #13074
- Feature: Remove option.fPIC for shared in **`conan new`** templates. #13066
- Feature: Add *conan cache clean* subcommand to clean build and source folders. #13050
- Feature: Implement customizable `CMakeToolchain.presets_prefix` so presets name prepend this. #13015
- Feature: Add *[system_tools]* section to profiles to use your own installed tools instead of the packages declared in the requires. #10166
- Fix: Fixes in powershell escaping. #13084
- Fix: Define `CMakeToolchain.presets_prefix="conan"` by default, to avoid conflict with other users presets. #13015

9.4 2.0.0-beta9 (31-Jan-2023)

- Feature: Add package names in Conan cache hash paths. #13011
- Feature: Implement `tools.build:download_source` conf to force the installation of sources in **`conan install`** or `conan graph info`. #13003
- Feature: Users can define their own settings in *settings_user.yml* that will be merged with the Conan *settings.yml*. #12980
- Feature: List disabled remotes too. #12937
- Fix: PkgConfiDeps is using the wrong `dependencies.host` from dependencies instead of `get_transitive_requires()` computation. #13013
- Fix: Fixing transitive shared linux libraries in CMakeDeps. #13010
- Fix: Fixing issues with test_package output folder. #12992
- Fix: Improve error messages for wrong methods. #12962
- Fix: Fix fail in parallel packages download due to database concurrency issues. #12930
- Fix: Enable authentication against disabled remotes. #12913
- Fix: Improving system_requirements. #12912
- Fix: Change tar format to PAX, which is the Python3.8 default. #12899

9.5 2.0.0-beta8 (12-Jan-2023)

- Feature: Add *unix_path_package_info_legacy* function for those cases in which it is used in *package_info* in recipes that require compatibility with Conan 1.x. In Conan 2, path conversions should not be performed in the *package_info* method. #12886
- Feature: New serialization json and printing for `conan list`. #12883
- Feature: Add requirements to *conan new cmake_{lib,exe}* #12875
- Feature: Allow `--no-remotes` to force temporal disabling of remotes #12808
- Feature: Add barebones template option to conan new. #12802
- Feature: Avoid requesting package configuration if PkgID is passed. #12801
- Feature: Implemented *conan list *#latest* and *conan list *:*#latest*. Basically, this command can show the latest RREVs and PREVs for all the matching references. #12781
- Feature: Allow chaining of *self.output* write methods #12780
- Fix: Make `graph info` filters to work on json output too #12836
- Bugfix: Fix bug to pass a valid GNU triplet when using AutotoolsToolchain and cross-building on Windows. #12881
- Bugfix: Ordering if same ref.name but different versions. #12801

9.6 2.0.0-beta7 (22-Dec-2022)

- Feature: Raise an error when a generator is both defined in generators attribute and instantiated in generate() method #12722
- Feature: *test_requires* improvements, including allowing it in conanfile.txt #12699
- Feature: Improve errors for when required_conan_version has spaces between the operator and the version #12695
- Feature: ConanAPI cleanup and organization #12666

9.7 2.0.0-beta6 (02-Dec-2022)

- Feature: Use `--confirm` to not request confirmation when removing instead of `--force` #12636
- Feature: Simplify loading conaninfo.txt for search results #12616
- Feature: Renamed ConanAPIV2 to ConanAPI #12615
- Feature: Refactor ConanAPI #12615
- Feature: Improve conan cache path command #12554
- Feature: Improve #latest and pattern selection from remove/upload/download #12572
- Feature: Add build_modules to provided deprecated warning to allow migration from 1.x #12578
- Feature: Lockfiles alias support #12525

9.8 2.0.0-beta5 (11-Nov-2022)

- Feature: Improvements in the remotes management and API #12468
- Feature: Implement env_info and user_info as fake attributes in Conan 2.0 #12351
- Feature: Improve settings.rm_safe() #12379
- Feature: New RecipeReference equality #12506
- Feature: Simplifying compress and uncompress of .tgz files #12378
- Feature: conan source command does not require a default profile #12475
- Feature: Created a proper LockfileAPI, with detailed methods (update, save, etc), instead of several loose methods #12502
- Feature: The conan export can also produce lockfiles, necessary for users doing a 2 step (export + install–build) process #12502
- Feature: Drop compat_app #12484
- Fix: Fix transitive propagation of transitive_headers=True #12508
- Fix: Fix transitive propagation of transitive_libs=False for static libraries #12508
- Fix: Fix test_package for python_requires #12508

9.9 2.0.0-beta4 (11-Oct-2022)

- Feature: Do not allow doing conan create/export with uncommitted changes using revision_mode=scm #12267
- Feature: Simplify conan inspect command, removing path subcommand #12263
- Feature: Add –deploy argument to graph info command #12243
- Feature: Pass graph object to deployers instead of ConanFile #12243
- Feature: Add included_files method to conan.tools.scm.Git #12246
- Feature: Improve detection of clang libcxx #12251
- Feature: Remove old profile variables system in favor of Jinja2 syntax in profiles #12152
- Fix: Update command to follow Conan 2.0 conventions about CLI output #12235
- Fix: Fix aggregation of test trait in diamonds #12080

9.10 2.0.0-beta3 (12-Sept-2022)

- Feature: Decouple test_package from create. #12046
- Feature: Warn if special chars in exported refs. #12053
- Feature: Improvements in MSBuildDeps traits. #12032
- Feature: Added support for CLICOLOR_FORCE env var, that will activate the colors in the output if the value is declared and different to 0. #12028
- Fix: Call source() just once for all configurations. #12050
- Fix: Fix deployers not creating output_folder. #11977
- Fix: Fix build_id() removal of require. #12019
- Fix: If Conan fails to load a custom command now it fails with a useful error message. #11720
- Bugfix: If the 'os' is not specified in the build profile and a recipe, in Windows, wanted to run a command. #11728

9.11 2.0.0-beta2 (27-Jul-2022)

- Feature: Add traits support in MSBuildDeps. #11680
- Feature: Add traits support in XcodeDeps. #11615
- Feature: Let dependency define package_id modes. #
- Feature: Add `conan.conanrc` file to setup the conan user home. #11675
- Feature: Add `core.cache:storage_path` to declare the absolute path where you want to store the Conan packages. #11672
- Feature: Add tools for checking max cppstd version. #11610
- Feature: Add a `post_build_fail` hook that is called when a build fails. #11593
- Feature: Add `pre_generate` and `post_generate` hook, covering the generation of files around the `generate()` method call. #11593

- Feature: Brought `conan config list` command back and other conf improvements. #11575
- Feature: Added two new arguments for all commands -v for controlling the verbosity of the output and –logger to output the contents in a json log format for log processors. #11522

9.12 2.0.0-beta1 (20-Jun-2022)

- Feature: New graph model to better support C and C++ binaries relationships, compilation, and linkage.
- Feature: New documented public Python API, for user automation
- Feature: New build system integrations, more flexible and powerful, and providing transparent integration when possible, like `CMakeDeps` and `CMakeToolchain`
- Feature: New custom user commands, that can be built using the public PythonAPI and can be shared and installed with `conan config install`
- Feature: New CLI interface, with cleaner commands and more structured output
- Feature: New deployers mechanism to copy artifacts from the cache to user folders, and consume those copies while building.
- Feature: Improved `package_id` computation, taking into account the new more detailed graph model.
- Feature: Added `compatibility.py` extension mechanism to allow users to define binary compatibility globally.
- Feature: Simpler and more powerful `lockfiles` to provide reproducibility over time.
- Feature: Better configuration with `[conf]` and better environment management with the new `conan.tools.env` tools.
- Feature: Conan cache now can store multiple revisions simultaneously.
- Feature: New extensions plugins to implement profile checking, package signing, and build commands wrapping.
- Feature: Used the package immutability for an improved update, install and upload flows.

INDEX

D

E

F

G

I

L

M

N

O

P

R

S

T

U

V

W

X

Y

Z