## CONTENTS

1 Introduction 3
   1.1 Open Source .......................................................... 3
   1.2 Decentralized package manager ..................................... 3
   1.3 Binary management .................................................. 4
   1.4 Cross platform, build system agnostic ............................. 4
   1.5 Stable ................................................................. 5

2 Install 7
   2.1 Install with pip (recommended) .................................... 7
   2.2 Install from brew (OSX) ............................................. 8
   2.3 Install from AUR (Arch Linux) ..................................... 8
   2.4 Install the binaries .................................................. 9
   2.5 Initial configuration ............................................... 9
   2.6 Install from source ................................................ 9
   2.7 Update .............................................................. 10
   2.8 Python 2 Deprecation Notice ..................................... 10

3 Getting Started 11
   3.1 An MD5 Encrypter using the Poco Libraries ........................ 11
   3.2 Installing Dependencies ........................................... 14
   3.3 Inspecting Dependencies .......................................... 15
   3.4 Searching Packages .............................................. 18
   3.5 Building with Other Configurations ............................... 18

4 Using packages 21
   4.1 Installing dependencies ............................................ 21
   4.2 Using profiles ..................................................... 26
   4.3 Workflows ......................................................... 27

5 Creating Packages 31
   5.1 Getting Started .................................................... 31
   5.2 Recipe and Sources in a Different Repo ............................. 35
   5.3 Recipe and Sources in the Same Repo ............................... 36
   5.4 Packaging Existing Binaries ...................................... 39
   5.5 Understanding Packaging ......................................... 41
   5.6 Defining Package ABI Compatibility .............................. 43
   5.7 Inspecting Packages ............................................... 51
   5.8 Packaging Approaches ............................................ 52
   5.9 Package Creator Tools ............................................ 57

6 Uploading Packages 59
<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>17.18</td>
<td>1.12.0 (30-Jan-2019)</td>
</tr>
<tr>
<td>17.19</td>
<td>1.11.2 (8-Jan-2019)</td>
</tr>
<tr>
<td>17.20</td>
<td>1.11.1 (20-Dec-2018)</td>
</tr>
<tr>
<td>17.21</td>
<td>1.11.0 (19-Dec-2018)</td>
</tr>
<tr>
<td>17.22</td>
<td>1.10.2 (17-Dec-2018)</td>
</tr>
<tr>
<td>17.23</td>
<td>1.10.1 (11-Dec-2018)</td>
</tr>
<tr>
<td>17.24</td>
<td>1.10.0 (4-Dec-2018)</td>
</tr>
<tr>
<td>17.25</td>
<td>1.9.2 (20-Nov-2018)</td>
</tr>
<tr>
<td>17.26</td>
<td>1.9.1 (08-Nov-2018)</td>
</tr>
<tr>
<td>17.27</td>
<td>1.9.0 (30-October-2018)</td>
</tr>
<tr>
<td>17.28</td>
<td>1.8.4 (19-October-2018)</td>
</tr>
<tr>
<td>17.29</td>
<td>1.8.3 (17-October-2018)</td>
</tr>
<tr>
<td>17.30</td>
<td>1.8.2 (10-October-2018)</td>
</tr>
<tr>
<td>17.31</td>
<td>1.8.1 (10-October-2018)</td>
</tr>
<tr>
<td>17.32</td>
<td>1.8.0 (9-October-2018)</td>
</tr>
<tr>
<td>17.33</td>
<td>1.7.4 (18-September-2018)</td>
</tr>
<tr>
<td>17.34</td>
<td>1.7.3 (6-September-2018)</td>
</tr>
<tr>
<td>17.35</td>
<td>1.7.2 (4-September-2018)</td>
</tr>
<tr>
<td>17.36</td>
<td>1.7.1 (31-August-2018)</td>
</tr>
<tr>
<td>17.37</td>
<td>1.7.0 (29-August-2018)</td>
</tr>
<tr>
<td>17.38</td>
<td>1.6.1 (27-July-2018)</td>
</tr>
<tr>
<td>17.39</td>
<td>1.6.0 (19-July-2018)</td>
</tr>
<tr>
<td>17.40</td>
<td>1.5.2 (5-July-2018)</td>
</tr>
<tr>
<td>17.41</td>
<td>1.5.1 (29-June-2018)</td>
</tr>
<tr>
<td>17.42</td>
<td>1.5.0 (27-June-2018)</td>
</tr>
<tr>
<td>17.43</td>
<td>1.4.5 (22-June-2018)</td>
</tr>
<tr>
<td>17.44</td>
<td>1.4.4 (11-June-2018)</td>
</tr>
<tr>
<td>17.45</td>
<td>1.4.3 (6-June-2018)</td>
</tr>
<tr>
<td>17.46</td>
<td>1.4.2 (4-June-2018)</td>
</tr>
<tr>
<td>17.47</td>
<td>1.4.1 (31-May-2018)</td>
</tr>
<tr>
<td>17.48</td>
<td>1.4.0 (30-May-2018)</td>
</tr>
<tr>
<td>17.49</td>
<td>1.3.3 (10-May-2018)</td>
</tr>
<tr>
<td>17.50</td>
<td>1.3.2 (7-May-2018)</td>
</tr>
<tr>
<td>17.51</td>
<td>1.3.1 (3-May-2018)</td>
</tr>
<tr>
<td>17.52</td>
<td>1.3.0 (30-April-2018)</td>
</tr>
<tr>
<td>17.53</td>
<td>1.2.3 (10-Apr-2017)</td>
</tr>
<tr>
<td>17.54</td>
<td>1.2.1 (3-Apr-2018)</td>
</tr>
<tr>
<td>17.55</td>
<td>1.2.0 (28-Mar-2018)</td>
</tr>
<tr>
<td>17.56</td>
<td>1.1.1 (5-Mar-2018)</td>
</tr>
<tr>
<td>17.57</td>
<td>1.1.0 (27-Feb-2018)</td>
</tr>
<tr>
<td>17.58</td>
<td>1.0.4 (30-January-2018)</td>
</tr>
<tr>
<td>17.59</td>
<td>1.0.3 (22-January-2018)</td>
</tr>
<tr>
<td>17.60</td>
<td>1.0.2 (16-January-2018)</td>
</tr>
<tr>
<td>17.61</td>
<td>1.0.1 (12-January-2018)</td>
</tr>
<tr>
<td>17.62</td>
<td>1.0.0 (10-January-2018)</td>
</tr>
<tr>
<td>17.63</td>
<td>1.0.0-beta5 (8-January-2018)</td>
</tr>
<tr>
<td>17.64</td>
<td>1.0.0-beta4 (4-January-2018)</td>
</tr>
<tr>
<td>17.65</td>
<td>1.0.0-beta3 (28-December-2017)</td>
</tr>
<tr>
<td>17.66</td>
<td>1.0.0-beta2 (23-December-2017)</td>
</tr>
<tr>
<td>17.67</td>
<td>0.30.3 (15-December-2017)</td>
</tr>
<tr>
<td>17.68</td>
<td>0.30.2 (14-December-2017)</td>
</tr>
<tr>
<td>17.69</td>
<td>0.30.1 (12-December-2017)</td>
</tr>
<tr>
<td>17.70</td>
<td>0.29.2 (2-December-2017)</td>
</tr>
<tr>
<td>17.71</td>
<td>0.29.1 (23-November-2017)</td>
</tr>
<tr>
<td>Version</td>
<td>Date</td>
</tr>
<tr>
<td>---------</td>
<td>-----------------</td>
</tr>
<tr>
<td>17.72</td>
<td>21-November-2017</td>
</tr>
<tr>
<td>17.73</td>
<td>31-October-2017</td>
</tr>
<tr>
<td>17.74</td>
<td>26-October-2017</td>
</tr>
<tr>
<td>17.75</td>
<td>20-September-2017</td>
</tr>
<tr>
<td>17.76</td>
<td>05-September-2017</td>
</tr>
<tr>
<td>17.77</td>
<td>31-August-2017</td>
</tr>
<tr>
<td>17.78</td>
<td>20-July-2017</td>
</tr>
<tr>
<td>17.79</td>
<td>19-July-2017</td>
</tr>
<tr>
<td>17.80</td>
<td>15-June-2017</td>
</tr>
<tr>
<td>17.81</td>
<td>05-June-2017</td>
</tr>
<tr>
<td>17.82</td>
<td>01-June-2017</td>
</tr>
<tr>
<td>17.83</td>
<td>03-May-2017</td>
</tr>
<tr>
<td>17.84</td>
<td>20-April-2017</td>
</tr>
<tr>
<td>17.85</td>
<td>18-April-2017</td>
</tr>
<tr>
<td>17.86</td>
<td>18-April-2017</td>
</tr>
<tr>
<td>17.87</td>
<td>04-April-2017</td>
</tr>
<tr>
<td>17.88</td>
<td>23-March-2017</td>
</tr>
<tr>
<td>17.89</td>
<td>21-March-2017</td>
</tr>
<tr>
<td>17.90</td>
<td>06-March-2017</td>
</tr>
<tr>
<td>17.91</td>
<td>02-March-2017</td>
</tr>
<tr>
<td>17.92</td>
<td>01-March-2017</td>
</tr>
<tr>
<td>17.93</td>
<td>27-February-2017</td>
</tr>
<tr>
<td>17.94</td>
<td>27-February-2017</td>
</tr>
<tr>
<td>17.95</td>
<td>15-February-2017</td>
</tr>
<tr>
<td>17.96</td>
<td>02-February-2017</td>
</tr>
<tr>
<td>17.97</td>
<td>31-January-2017</td>
</tr>
<tr>
<td>17.98</td>
<td>11-January-2017</td>
</tr>
<tr>
<td>17.99</td>
<td>03-January-2017</td>
</tr>
<tr>
<td>17.100</td>
<td>21-December-2016</td>
</tr>
<tr>
<td>17.101</td>
<td>15-December-2016</td>
</tr>
<tr>
<td>17.102</td>
<td>13-December-2016</td>
</tr>
<tr>
<td>17.103</td>
<td>05-December-2016</td>
</tr>
<tr>
<td>17.104</td>
<td>19-November-2016</td>
</tr>
<tr>
<td>17.105</td>
<td>08-November-2016</td>
</tr>
<tr>
<td>17.106</td>
<td>20-October-2016</td>
</tr>
<tr>
<td>17.107</td>
<td>20-October-2016</td>
</tr>
<tr>
<td>17.108</td>
<td>13-October-2016</td>
</tr>
<tr>
<td>17.109</td>
<td>03-October-2016</td>
</tr>
<tr>
<td>17.110</td>
<td>13-September-2016</td>
</tr>
<tr>
<td>17.111</td>
<td>31-August-2016</td>
</tr>
<tr>
<td>17.112</td>
<td>3-August-2016</td>
</tr>
<tr>
<td>17.113</td>
<td>29-June-2016</td>
</tr>
<tr>
<td>17.114</td>
<td>11-May-2016</td>
</tr>
<tr>
<td>17.115</td>
<td>3-May-2016</td>
</tr>
<tr>
<td>17.116</td>
<td>28-Mar-2016</td>
</tr>
<tr>
<td>17.117</td>
<td>15-Mar-2016</td>
</tr>
<tr>
<td>17.118</td>
<td>5-Feb-2016</td>
</tr>
<tr>
<td>17.119</td>
<td>11-Jan-2016</td>
</tr>
<tr>
<td>17.120</td>
<td>18-Dec-2015</td>
</tr>
</tbody>
</table>
Conan is a portable package manager, intended for C and C++ developers, but it is able to manage builds from source, dependencies, and precompiled binaries for any language.

For more information, check conan.io.

Contents:
1.1 Open Source

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

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.

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

The different applications in the image above are:

- The **Conan client**: this is a console/terminal command line application, containing the heavy logic for package creation and consumption. Conan client has a local cache for package storage, and so it allows you to fully create and test packages offline. You can also **work offline** so long as no new packages are needed from remote servers.

- The **Conan server**: this is a TCP server that can be easily run as your **own server on-premises** to host your private packages. It is also a service application that can be run as a daemon or service, behind a web server (apache, nginx), or easily as stand-alone application. Both the Conan client and the conan_server are OSS, MIT license, so you can use them for free in your company, customize them, or redistribute them without any legal issue.
• **JFrog Artifactory** offers Conan repositories; so it can also be used as an on-premises server. It is a more powerful solution, featuring a WebUI, multiple auth protocols, High Availability, etc. It also has cloud offerings that will allow you to have private packages without having any on-premises infrastructure.

• **JFrog Bintray** provides a public and free hosting service for OSS Conan packages. Users can create their own repositories under their accounts and organizations, and freely upload Conan packages there, without moderation. You should, however, take into account that those packages will be public, and so they must conform to the respective licenses, especially if the packages contain third party code. Just reading or retrieving Conan packages from Bintray, doesn’t require an account, an account is only needed to upload packages. Besides that, Bintray provides a central repository called **conan-center** which is moderated, and packages are reviewed before being accepted to ensure quality.

### 1.3 Binary management

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

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

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

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

### 1.4 Cross platform, build system agnostic

Conan works and is being actively used on Windows, Linux (Ubuntu, Debian, RedHat, ArchLinux, Raspbian), OSX, FreeBSD, and SunOS, and, as it is portable, it might work in any other platform that can run python. In the documentation, examples for a specific OS might be found, such as `conan install . -s compiler="Visual`
Studio", which will be specific for Windows users. If on a different system, the reader should adapt to their own platform and settings (for example `conan install . -s compiler=gcc`).

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

### 1.5 Stable

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

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

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

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

Additionally, starting in version 1.6, we began the process of deprecating Python2 support. Features already working with python2 will continue to do so, but new ones may require Python3. See the deprecation notice for more details.

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.

Have any questions? Please check out our FAQ section or.
Conan can be installed in many Operating Systems. It has been extensively used and tested in Windows, Linux (different distros), OSX, and is also actively used in FreeBSD and Solaris SunOS. There are also several additional operating systems on which it has been reported to work.

There are three ways to install Conan:

1. The preferred and strongly recommended way to install Conan is from PyPI, the Python Package Index, using the `pip` command.

2. There are other available installers for different systems, which might come with a bundled python interpreter, so that you don’t have to install python first. Note that some of these installers might have some limitations, specially those created with pyinstaller (such as Windows exe & Linux deb).

3. Running Conan from sources.

### 2.1 Install with pip (recommended)

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

**Warning:** Python 2 will soon be deprecated by the Python maintainers. It is strongly recommended to use Python 3 with Conan, especially if need to manage non-ascii filenames or file contents. Conan still supports Python 2, however some of the dependencies have started to be supported only by Python 3. See [Python 2 Deprecation Notice](https://docs.python.org/2/whatsnew/2.7.html) for details.

Install Conan:

```
$ pip install conan
```

**Important: Please READ carefully**

- Make sure that your pip installation matches your Python (2.7 or 3.X) 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) If not using a virtualenv it is possible that conan dependencies will conflict with previously existing dependencies, specially if you are using Python for other purposes.
• In Windows and Python 2.7, you may need to use 32bit python distribution (which is the Windows default), instead of 64 bit.

• 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.

• If you are using Windows and Python <3.5, you may have issues if Python is installed in a path with spaces, such as “C:/Program Files(x86)/Python”. This is a known Python limitation, and is not related to Conan. Try installing Python in a path without spaces, use a virtualenv in another location or upgrade your Python installation.

• 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.

• Windows, Python 3 installation can fail installing the wrapt dependency because of a bug in pip. Information about this issue and workarounds is available here: https://github.com/GrahamDumpleton/wrapt/issues/112.

• Conan works with Python 2.7, but not all features are available when not using Python 3.x starting with version 1.6

2.1.1 Known installation issues with pip

• With Python 3.4 the installation sometimes fails with NameError: name 'implementation_name' is not defined. This issue seems to happen when using pip without a virtual environment. You can fix it by upgrading pip this way:

```bash
$ curl https://bootstrap.pypa.io/get-pip.py -o get-pip.py
$ python3.4 get-pip.py
$ pip install conan
```

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

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

2.2 Install from brew (OSX)

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

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

2.3 Install from AUR (Arch Linux)

The easiest way to install Conan on Arch Linux is by using one of the Arch User Repository (AUR) helpers, e.g., yay, aurman, or pakku. For example, the following command installs Conan using yay:

```bash
$ yay -S conan
```
Alternatively, build and install Conan manually using `makepkg` and `pacman` as described in the Arch Wiki. Conan build files can be downloaded from AUR: https://aur.archlinux.org/packages/conan/. Make sure to first install the three Conan dependencies which are also found in AUR:

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

## 2.4 Install the binaries

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

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

## 2.5 Initial configuration

Check if Conan is installed correctly. Run the following command in your console:

```bash
$ conan
```

The response should be similar to:

<table>
<thead>
<tr>
<th>Consumer commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>install</td>
</tr>
<tr>
<td>config</td>
</tr>
<tr>
<td>get</td>
</tr>
<tr>
<td>info</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

## 2.6 Install from source

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

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

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

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

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

import sys

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

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

$ conan

You should see the Conan commands help.

## 2.7 Update

If installed via pip, Conan can be easily updated:

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

If installed via the installers (.exe, .deb), download the new installer and execute it.

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

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

## 2.8 Python 2 Deprecation Notice

All features of Conan until version 1.6 are fully supported in both Python 2 and Python 3. However, new features in upcoming Conan releases that are only available in Python 3 or more easily available in Python 3 will be implemented and tested only in Python 3, and versions of Conan using Python 2 will not have access to that feature. This will be clearly described in code and documentation.

If and when Conan 2.x is released (Not expected in 2018) the level of compatibility with Python 2 may be reduced further.

We encourage you to upgrade to Python 3 as soon as possible. However, if this is impossible for you or your team, we would like to know it. Please give feedback in the Conan issue tracker or write us to info@conan.io.
Let's get started with an example: We are going to create an MD5 encrypter app that uses one of the most popular C++ libraries: Poco.

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.

### 3.1 An MD5 Encrypter using the Poco Libraries

**Note:** The source files to recreate this project are available in the following GitHub repository. You can skip the manual creation of folder and sources with this command:

```bash
$ git clone https://github.com/conan-io/examples.git && cd examples/libraries/poco/md5
```

1. Create the following source file inside a folder. This will be the source file of our application:

   **Listing 1: md5.cpp**

   ```cpp
   #include "Poco/MD5Engine.h"
   #include "Poco/DigestStream.h"

   #include <iostream>

   int main(int argc, char** argv)
   {
     Poco::MD5Engine md5;
     Poco::DigestOutputStream ds(md5);
     ds << "abcdefghijklmnopqrstuvwxyz"
     .close();
     std::cout << Poco::DigestEngine::digestToHex(md5.digest()):
     ↪std::endl;
     return 0;
   }
   ```

2. We know that our application relies on the Poco libraries. Let's look for it in the Conan Center remote:

   ```bash
   $ conan search Poco* --remote=conan-center
   Existing package recipes:

   Poco/1.7.8p3@pocoproject/stable
   ```

   (continues on next page)
3. We got some interesting references for Poco. Let’s inspect the metadata of the 1.9.0 version:

```bash
$ conan inspect Poco/1.9.0@pocoproject/stable
...
name: Poco
version: 1.9.0
url: http://github.com/pocoproject/conan-poco
license: The Boost Software License 1.0
author: None
description: Modern, powerful open source C++ class libraries for building network- and internet-based applications that run on desktop, server, mobile and embedded systems.
generators: ('cmake', 'txt')
exports: None
exports_sources: ('CMakeLists.txt', 'PocoMacros.cmake')
short_paths: False
apply_env: True
build_policy: None
settings: ('os', 'arch', 'compiler', 'build_type')
options:
   enable_apacheconnector: [True, False]
   shared: [True, False]
default_options:
   enable_apacheconnector: False
   shared: False
```

4. Ok, it looks like this dependency could work with our Encrypter app. We should indicate which are the requirements and the generator for our build system. Let’s create a `conanfile.txt` inside our project’s folder with the following content:

```plaintext
Listing 2: conanfile.txt

[requires]
Poco/1.9.0@pocoproject/stable

[generators]
cmake
```

In this example we are using CMake to build the project, which is why the cmake generator is specified. This generator creates a `conanbuildinfo.cmake` file that defines CMake variables including paths and library names that can be used in our build. Read more about Generators.

5. Next step: We are going to install the required dependencies and generate the information for the build system:

**Important:** If you are using GCC compiler >= 5.1, Conan will set the compiler.libcxx to the old ABI for backwards compatibility. You can change this with the following commands:
$ conan profile new default --detect  # Generates default profile
→ detecting GCC and sets old ABI
$ conan profile update settings.compiler.libcxx=libstdc++11 default  #
→ Sets libcxx to C++11 ABI

You will find more information in *How to manage the GCC >= 5 ABI*.

$ mkdir build && cd build
$ conan install ..
...
Requirements
  OpenSSL/1.0.2o@conan/stable from 'conan-center' - Downloaded
  Poco/1.9.0@pocoproject/stable from 'conan-center' - Cache
  zlib/1.2.11@conan/stable from 'conan-center' - Downloaded
Packages
  OpenSSL/1.0.2o@conan/stable:606fdb601e335c2001bdf31d47826b644747077 -
→ Download
  Poco/1.9.0@pocoproject/
→ stable:09378ed7f5118538e9f04b212b79fe2d12d005c - Download
  zlib/1.2.11@conan/stable:6cc50b139b9c3d27b3e9042d5f5372d327b3a9f7 -
→ Download
zlib/1.2.11@conan/stable: Retrieving package,
→ 6cc50b139b9c3d27b3e9042d5f5372d327b3a9f7 from remote 'conan-center'
...
Download conan_package.tgz
[================================================================] 99.8KB/99.8KB
...
zlib/1.2.11@conan/stable: Package installed,
→ 6cc50b139b9c3d27b3e9042d5f5372d327b3a9f7
OpenSSL/1.0.2o@conan/stable: Retrieving package,
→ 606fdb601e335c2001bdf31d47826b644747077 from remote 'conan-center'
...
Download conan_package.tgz
[================================================================] 5.5MB/5.5MB
...
OpenSSL/1.0.2o@conan/stable: Package installed,
→ 606fdb601e335c2001bdf31d47826b644747077
Poco/1.9.0@pocoproject/stable: Retrieving package,
→ 09378ed7f5118538e9f04b212b79fe2d12d005c from remote 'conan-center'
...
Download conan_package.tgz
[================================================================] 11.5MB/11.5MB
...
Poco/1.9.0@pocoproject/stable: Package installed,
→ 09378ed7f5118538e9f04b212b79fe2d12d005c
PROJECT: Generator cmake created conanbuildinfo.cmake
PROJECT: Generator txt created conanbuildinfo.txt
PROJECT: Generated conaninfo.txt

Conan installed our Poco dependency but also the **transitive dependencies** for it: OpenSSL and zlib.
It has also generated a *conanbuildinfo.cmake* file for our build system.

6. Now let’s create our build file. To inject the Conan information, include the generated *conanbuildinfo.cmake* file like this:
Listing 3: CMakeLists.txt

```cmake
cmake_minimum_required(VERSION 2.8.12)
project(MD5Encrypter)
add_definitions("-std=c++11")
include(${CMAKE_BINARY_DIR}/conanbuildinfo.cmake)
conan_basic_setup()
add_executable(md5 md5.cpp)
target_link_libraries(md5 ${CONAN_LIBS})
```

7. Now we are ready to build and run our Encrypter app:

```bash
(win)
$ cmake .. -G "Visual Studio 15 Win64"
$ cmake --build . --config Release

(linux, mac)
$ cmake .. -G "Unix Makefiles" -DCMAKE_BUILD_TYPE=Release
$ cmake --build .
{100%} Built target md5
$ ./bin/md5
c3fcd3d76192e4007dfb496cca67e13b
```

### 3.2 Installing Dependencies

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

**Note:** Conan generates a default profile with your detected settings (OS, compiler, architecture…) and that configuration is printed at the top of every `conan install` command. However, it is strongly recommended to review it and adjust the settings to accurately describe your system as shown in the Building with Other Configurations section.

It is very important to understand the installation process. When the `conan install` command runs, settings specified on the command line or taken from the defaults in `<userhome>/conan/profiles/default` file are applied.
For example, the command `conan install . --settings os="Linux" --settings compiler="gcc"`, performs these steps:

- Checks if the package recipe (for Poco/1.9.0@pocoproject/stable package) exists in the local cache. If we are just starting, the cache is empty.

- Looks for the package recipe in the defined remotes. Conan comes with conan-center Bintray remote as the default, but can be changed.

- If the recipe exists, the Conan client fetches and stores it in your local cache.

- With the package recipe and the input settings (Linux, GCC), Conan looks for the corresponding binary in the local cache.

- Then Conan searches the corresponding binary package in the remote and fetches it.

- Finally, it generates an appropriate file for the build system specified in the `[generators]` section.

There are binaries for several mainstream compilers and versions available in Conan Center repository in Bintray, such as Visual Studio 14, 15, Linux GCC 4.9 and Apple Clang 3.5... Conan will throw an error if the binary package required for specific settings doesn’t exist. You can build the binary package from sources using `conan install --build=missing`, it will succeed if your configuration is supported by the recipe. You will find more info in the `Building with Other Configurations` section.

### 3.3 Inspecting Dependencies

The retrieved packages are installed to your local user cache (typically `.conan/data`), and can be reused from this location for other projects. This allows to clean your current project and continue working even without network connection. To search for packages in the local cache run:

For example, the command `conan install . --settings os="Linux" --settings compiler="gcc"`, performs these steps:

- Checks if the package recipe (for Poco/1.9.0@pocoproject/stable package) exists in the local cache. If we are just starting, the cache is empty.

- Looks for the package recipe in the defined remotes. Conan comes with conan-center Bintray remote as the default, but can be changed.

- If the recipe exists, the Conan client fetches and stores it in your local cache.

- With the package recipe and the input settings (Linux, GCC), Conan looks for the corresponding binary in the local cache.

- Then Conan searches the corresponding binary package in the remote and fetches it.

- Finally, it generates an appropriate file for the build system specified in the `[generators]` section.

There are binaries for several mainstream compilers and versions available in Conan Center repository in Bintray, such as Visual Studio 14, 15, Linux GCC 4.9 and Apple Clang 3.5... Conan will throw an error if the binary package required for specific settings doesn’t exist. You can build the binary package from sources using `conan install --build=missing`, it will succeed if your configuration is supported by the recipe. You will find more info in the `Building with Other Configurations` section.

### 3.3. Inspecting Dependencies

The retrieved packages are installed to your local user cache (typically `.conan/data`), and can be reused from this location for other projects. This allows to clean your current project and continue working even without network connection. To search for packages in the local cache run:
To inspect the different binary packages of a reference run:

```
$ conan search Poco/1.9.0@pocoproject/stable
Existing packages for recipe Poco/1.9.0@pocoproject/stable:

Package_ID: 09378ed7f51185386e9f04b212b79fe2d12d005c
  [options]
      cxx_14: False
    enable_apacheconnector: False
      enable_cppparser: False
        enable_crypto: True
      enable_data: True
...
```

There is also the possibility to generate a table for all package binaries available in a remote:

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

To inspect all your current project’s dependencies use the `conan info` command by pointing it to the location of the `conanfile.txt` folder:

```
$ conan info ..
PROJECT
  ID: 6ecacba4f2b7535e0acb633a0cc4de0234445fea
  BuildID: None
  Requires:
    Poco/1.9.0@pocoproject/stable
OpenSSL/1.0.2o@conan/stable
  ID: 606fdb601e335c2001bdf31d478826b644747077
  BuildID: None
  Remote: conan-center=https://conan.bintray.com
```

(continues on next page)
URL: http://github.com/conan-community/conan-openssl
License: The current OpenSSL licence is an 'Apache style' license: https://www.openssl.org/source/license.html
Recipe: Cache
Binary: Cache
Binary remote: conan-center
Creation date: 2018-08-27 09:12:47
Required by:
  Poco/1.9.0@pocoproject/stable
Requires:
  zlib/1.2.11@conan/stable
Poco/1.9.0@pocoproject/stable
  ID: 09378ed7f51185386e9f04b212b79fe2d12d005c
  BuildID: None
  Remote: conan-center=https://conan.bintray.com
  URL: http://github.com/pocoproject/conan-poco
  License: The Boost Software License 1.0
  Recipe: Cache
  Binary: Cache
  Binary remote: conan-center
  Creation date: 2018-08-30 13:28:08
  Required by:
    PROJECT
    Requires:
      OpenSSL/1.0.2o@conan/stable
      zlib/1.2.11@conan/stable
  ID: 6cc50b139b9c3d27b3e9042d5f5372d327b3a9f7
  BuildID: None
  Remote: conan-center=https://conan.bintray.com
  URL: http://github.com/conan-community/conan-zlib
  License: Zlib
  Recipe: Cache
  Binary: Cache
  Binary remote: conan-center
  Creation date: 2018-10-24 12:40:49
  Required by:
    OpenSSL/1.0.2o@conan/stable
Or generate a graph of your dependencies using Dot or HTML formats:

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

3.3. Inspecting Dependencies
3.4 Searching Packages

The remote repository where packages are installed from is configured by default in Conan. It is called Conan Center (configured as conan-center remote) and it is located in Bintray.

You can search packages in Conan Center using this command:

```
$ conan search "*" --remote=conan-center
```

Existing package recipes:

- Assimp/4.1.0@jacmoe/stable
- CLI11/1.6.1@cliutils/stable
- CTRE/2.1@ctre/stable
- Catch/1.12.1@bincrafters/stable
- Expat/2.2.5@pix4d/stable
- FakeIt/2.0.5@gasuketsu/stable
- IlmBase/2.2.0@Mikayex/stable
- IrrXML/1.2@conan/stable
- OpenSSL/1.0.2@conan/stable
- ...

There are additional community repositories that can be configured and used. See Bintray Repositories for more information.

3.5 Building with Other Configurations

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

A profile needs to be available prior to running commands such as conan install. When running the command, your settings are automatically detected (compiler, architecture...) and stored as the default profile. You can edit these settings ~/.conan/profiles/default or create new profiles with your desired configuration.

For example, if we have a profile with a 32-bit GCC configuration in a profile called gcc_x86, we can run the following:

```
$ conan install . --profile=gcc_x86
```

**Tip:** We strongly recommend using Profiles and managing them with conan config install.

However, the user can always override the default profile settings in the conan install command using the --settings parameter. As an exercise, try building the Encrypter project 32-bit version:

```
$ conan install . --profile=gcc_x86 --settings arch=x86_64
```

The above command installs a different package, using the --settings arch=x86 instead of the one of the profile used previously.

To use the 32-bit binaries, you will also have to change your project build:

- In Windows, change the CMake invocation to Visual Studio 14.
- In Linux, you have to add the -m32 flag to your CMakeLists.txt by running
  ```
  SET(CMAKE_CXX_FLAGS "$\{CMAKE_CXX_FLAGS\} -m32"), and the same applies to
  CMAKE_C_FLAGS, CMAKE_SHARED_LINK_FLAGS and CMAKE_EXE_LINKER_FLAGS. This can
  also be done more easily, by automatically using Conan, as we’ll show later.
• In macOS, you need to add the definition `-DCMAKE_OSX_ARCHITECTURES=i386`.

Got any doubts? Check our FAQ, or join the community in Cpplang Slack #conan channel!
This section shows how to setup your project and manage dependencies (i.e., install existing packages) with Conan.

### 4.1 Installing dependencies

In *Getting started* we used the `conan install` command to download the Poco library and build an example. If you inspect the `conanbuildinfo.cmake` file that was created when running `conan install`, you can see there that there are many CMake variables declared. For example `CONAN_INCLUDE_DIRS_ZLIB`, that defines the include path to the zlib headers, and `CONAN_INCLUDE_DIRS` that defines include paths for all dependencies headers.

If you check the full path that each of these variables defines, you will see that it points to a folder under your `<userhome>` folder. Together, these folders are the *local cache*. This is where package recipes and binary packages are stored and cached, so they don’t have to be retrieved again. You can inspect the local cache with `conan search`, and remove packages from it with `conan remove` command.
If you navigate to the folders referenced in `conanbuildinfo.cmake` you will find the headers and libraries for each package.

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

### 4.1.1 Requires

The required dependencies should be specified in the `[requires]` section. Here is an example:

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

Where:

- `Poco` is the name of the package which is usually the same as the project/library.
- `1.9.0` is the version which usually matches that of the packaged project/library. This can be any string; it does not have to be a number, so, for example, it could indicate if this is a “develop” or “master” version. Packages can be overwritten, so it is also OK to have packages like “nightly” or “weekly”, that are regenerated periodically.
- `pocoproject` is the owner of this package. It is basically a namespace that allows different users to have their own packages for the same library with the same name.
- `stable` is the channel. Channels provide another way to have different variants of packages for the same library and use them interchangeably. They usually denote the maturity of the package as an arbitrary string such as “stable” or “testing”, but they can be used for any purpose such as package revisions (e.g., the library version has not changed, but the package recipe has evolved).

#### Overriding requirements

You can specify multiple requirements and **override** transitive “require’s requirements”. In our example, Conan installed the Poco package and all its requirements transitively:

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

**Tip:** This is a good example of overriding requirements given the importance of keeping the OpenSSL library updated.

Consider that a new release of the OpenSSL library has been released, and a new corresponding Conan package is available. In our example, we do not need to wait until pocoproject (the author) generates a new package of POCO that includes the new OpenSSL library.

We can simply enter the new version in `[requires]` section:

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

The second line will override the OpenSSL/1.0.2l required by POCO with the currently non-existent `OpenSSL/1.0.2p`.  

---

22 Chapter 4. Using packages
Another example in which we may want to try some new zlib alpha features, we could replace the zlib requirement with one from another user or channel.

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

**Note:** You can use environment variable `CONAN_ERROR_ON_OVERRIDE` to raise an error for every overridden requirement not marked explicitly with the `override` keyword.

### 4.1.2 Generators

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

For a full list of generators, please refer to the complete `generators` reference.

### 4.1.3 Options

We have already seen that there are some `settings` that can be specified during installation. For example, `conan install . -s build_type=Debug`. These settings are typically a project-wide configuration defined by the client machine, so they cannot have a default value in the recipe. For example, it doesn’t make sense for a package recipe to declare “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.

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

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

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

To see only specific fields of the recipe you can use the `conan inspect` command instead:

```
$ conan inspect Poco/1.9.0@pocoproject/stable -a=options
$ conan inspect Poco/1.9.0@pocoproject/stable -a=default_options
```

For example, we can modify the previous example to use dynamic linkage instead of the default one, which was static, by editing the `conanfile.txt`:

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

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

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

As an alternative to defining options in the `conanfile.txt` file, you can specify them directly in the command line:

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

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

Finally, launch the executable:

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

What happened? It fails because it can’t find the shared libraries in the path. Remember that shared libraries are used at runtime, so the operating system, which is running the application, must be able to locate them.

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

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

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

You can add the libraries’ folders to the path (dynamic linker LD_LIBRARY_PATH path in Linux, DYLD_LIBRARY_PATH in OSX, or system PATH in Windows), or copy those shared libraries to some system folder where they can be found by the OS. But these operations are are typical operations deployments or final installation of apps; they are not desired during development, and Conan is intended for developers, so it avoids manipulations on the OS.

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

This is done using the [imports] section in conanfile.txt.

To demonstrate this, edit the conanfile.txt file and paste the following [imports] section:

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

[generators]
cmake

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

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

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

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

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

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

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

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

**See also:**

To learn more about working with shared libraries, please refer to Howtos/Manage shared libraries.
4.2 Using profiles

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

However, in large projects, configurations can get complex, settings can be very different, and we need an easy way to switch between different configurations with different settings, options etc. An easy way to switch between configurations is by using profiles.

A profile file contains a predefined set of **settings**, **options**, **environment variables**, and **build_requires** specified in the following structure:

```
[settings]
setting=value

[options]
MyLib:shared=True

[env]
env_var=value

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

Options allow the use of wildcards letting you apply the same option value to many packages. For example:

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

Here is an example of a configuration that a profile file may contain:

```
Listing 1: clang_3.5

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

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

A profile file can be stored in the default profile folder, or anywhere else in your project file structure. To use the configuration specified in a profile file, pass in the file as a command line argument as shown in the example below:

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

Continuing with the example of Poco, instead of passing in a long list of command line arguments, we can define a handy profile that defines them all and pass that to the command line when installing the different project dependencies. A profile to install dependencies as **shared** and in **debug** mode would look like this:
4.3 Workflows

This section summarizes some possible layouts and workflows when using Conan together with other tools as an end-user for installing and consuming existing packages. To create your own packages, please refer to Creating Packages.

Whether you are working on a single configuration or a multi configuration project, in both cases, the recommended approach is to have a conanfile (either .py or .txt) at the root of your project.

4.3.1 Single configuration

When working with a single configuration, your conanfile will be quite simple as shown in the examples and tutorials we have used so far in this user guide. For example, in Getting started, we showed how you can run the conan install command.
install .. command inside the build folder resulting in the conaninfo.txt and conanbuildinfo.cmake files being generated there too. Note that the build folder is temporary, so you should exclude it from version control to exclude these temporary files.

Out-of-source builds are also supported. Let’s look at a simple example:

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

This will result in the following layout:

```
example-poco-build
    conaninfo.txt
    conanbuildinfo.txt
    conanbuildinfo.cmake
example-poco-timer
    CMakeLists.txt  # If using cmake, but can be Makefile, sln...
    LICENSE
    README.md
    conanfile.txt
    timer.cpp
```

Now you are ready to build:

```
$ cd example-poco-build
$ cmake ../example-poco-timer -G "Visual Studio 15 Win64"  # or other generator
$ cmake --build . --config Release
$ ./bin/timer
```

We have created a separate build configuration of the project without affecting the original source directory in any way. The benefit is that we can freely experiment with the configuration: We can clear the build folder and build another. For example, changing the build type to Debug:

```
$ rm -rf *
$ conan install ../example-poco-timer -s build_type=Debug
$ cmake ../example-poco-timer -G "Visual Studio 15 Win64"
$ cmake --build . --config Debug
$ ./bin/timer
```

### 4.3.2 Multi configuration

You can also manage different configurations, whether in-source or out of source, and switch between them without having to re-issue the conan install command (Note however, that even if you did have to run conan install again, since subsequent runs use the same parameters, they would be very fast since packages would already have been installed in the local cache rather than in the project)

```
$ git clone https://github.com/memsharded/example-poco-timer
$ conan install example-poco-timer -s build_type=Debug -if example-poco-build/debug
$ conan install example-poco-timer -s build_type=Release -if example-poco-build/
    --release
$ cd example-poco-build/debug && cmake ../example-poco-timer -G "Visual Studio 15
    -Win64" && cd .....
$ cd example-poco-build/release && cmake ../example-poco-timer -G "Visual Studio
    15 Win64" && cd .....
```
Note: You can either use the `--install-folder` or `-if` flags to specify where to generate the output files, or manually create the output directory and navigate to it before executing the `conan install` command.

So the layout will be:

```
example-poco-build
  debug
    conaninfo.txt
    conanbuildinfo.txt
    conanbuildinfo.cmake
    CMakeCache.txt # and other cmake files
  release
    conaninfo.txt
    conanbuildinfo.txt
    conanbuildinfo.cmake
    CMakeCache.txt # and other cmake files
example-poco-timer
  CMakeLists.txt  # If using cmake, but can be Makefile, sln...
  LICENSE
  README.md
  conanfile.txt
  timer.cpp
```

Now you can switch between your build configurations in exactly the same way you do for CMake or other build systems, by moving to the folder in which the build configuration is located, because the Conan configuration files for that build configuration will also be there.

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

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

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

See also:

There are two generators, `cmake_multi` and `visual_studio_multi` that could help to avoid the context switch and using Debug and Release configurations simultaneously. Read more about them in `cmake_multi` and `visual_studio_multi`
This section shows how to create, build and test your packages.

5.1 Getting Started

To start learning about creating packages, we will create a package from the existing source code repository: https://github.com/conan-io/hello. You can check that project, it is a very simple “hello world” C++ library, using CMake as the build system to build a library and an executable. It does not contain any association with Conan.

We are using a similar GitHub repository as an example, but the same process also applies to other source code origins, like downloading a zip or tarball from the internet.

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

5.1.1 Creating the Package Recipe

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

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

This will generate the following files:

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

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

Let’s have a look at the root package recipe conanfile.py:

```python
from conans import ConanFile, CMake, tools

class HelloConan(ConanFile):
```

(continues on next page)
This is a complete package recipe. Without going into detail, these are the basics:

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

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

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

(continues on next page)
Learn more in the *Cross building* section.

- This package recipe is also able to create different binary packages for static and shared libraries with the *shared* option, which is set by default to *False* (i.e. by default it will use static linkage).

- The *source()* method executes a *git clone* to retrieve the sources from Github. Other origins, such as downloading a zip file are also available. As you can see, any manipulation of the code can be done, such as checking out any branch or tag, or patching the source code. In this example, we are adding two lines to the existing CMake code, to ensure binary compatibility. Don’t worry about it now, we’ll deal with it later.

- The *build()* configures the project, and then proceeds to build it using standard CMake commands. The CMake object just assists to translate the Conan settings to CMake command line arguments. Please note that *CMake is not strictly required*. You can build packages directly by invoking *make*, *MSBuild*, *SCons* or any other build system.

**See also:**

- Check the *existing build helpers*.

- The *package()* method copies artifacts (headers, libs) from the build folder to the final package folder.

- Finally, the *package_info()* method defines that the consumer must link with the “hello” library when using this package. Other information as include or lib paths can be defined as well. This information is used for files created by generators to be used by consumers, as *conanbuildinfo.cmake*.

**Note:** When writing your own *conanfile.py* references, please bear in mind that you should follow the rules in *conanfile.py*.

### 5.1.2 The test_package Folder

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

If you look at the *test_package* folder, you will realize that the *example.cpp* and the *CMakeLists.txt* files don’t have unique characteristics. The *test_package/conanfile.py* file is just another recipe, that can be perceived as a consumer *conanfile.txt* that has been displayed in previous sections:

```python
from conans import ConanFile, CMake
import os

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

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

(continues on next page)
cmake.configure()
cmake.build()

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

def test(self):
    os.chdir("bin")
    self.run("%.@example" % os.sep)

The `conanfile.py` described above has the following characteristics:

- It doesn’t have a name and version, as we are not creating a package so they are not necessary.
- The `package()` and `package_info()` methods are not required since we are not creating a package.
- The `test()` method specifies which binaries need to run.
- The `imports()` method is set to copy the shared libraries to the `bin` folder. When dynamic linking is applied, and the `test()` method launches the `example` executable, they are found causing the `example` to run.

**Note:** An important difference with respect to standard package recipes is that you don’t have to declare a `requires` attribute to depend on the tested `Hello/0.1@demo/testing` package as the `requires` will automatically be injected by Conan during the run. However, if you choose to declare it explicitly, it will work, but you will have to remember to bump the version, and possibly also the user and channel if you decide to change them.

### 5.1.3 Creating and Testing Packages

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

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

If “Hello world!” is displayed, it worked.

The `conan create` command does the following:

- Copies (“export” in Conan terms) the `conanfile.py` from the user folder into the `local cache`.
- Installs the package, forcing it to be built from the sources.
- Moves to the `test_package` folder and creates a temporary `build` folder.
- Executes the `conan install` ..., to install the requirements of the `test_package/conanfile.py`. Note that it will build “Hello” from the sources.
- Builds and launches the `example` consuming application, calling the `test_package/conanfile.py build()` and `test()` methods respectively.

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

```
$ conan export . demo/testing
$ conan install Hello/0.1@demo/testing --build=Hello
# package is created now, use test to test it
$ conan test test_package Hello/0.1@demo/testing
```
The `conan create` command receives the same command line parameters as `conan install` so you can pass to it the same settings, options, and command line switches. If you want to create and test packages for different configurations, you could:

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

5.1.4 Settings vs. Options

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

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

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

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

On the other hand, **options** are a package-specific configuration. Static or shared library are not settings that apply to all packages. Some can be header only libraries while others packages can be just data, or package executables. Packages can contain a mixture of different artifacts. `shared` is a common option, but packages can define and use any options they want.

Options are defined in the package recipe, including their supported values, while other can be defaulted by the package recipe itself. A package for a library can well define that by default it will be a static library (a typical default). If not specified other, the package will be static.

There are some exceptions to the above. For example, settings can be defined per-package using the command line:

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

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

There are situations whereby many packages use the same option, thereby allowing you to set its value once using patterns, like:

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

Any doubts? Please check out our FAQ section or .

5.2 Recipe and Sources in a Different Repo

In the previous section, we fetched the sources of our library from an external repository. It is a typical workflow for packaging third party libraries.

There are two different ways to fetch the sources from an external repository:

1. Using the `source()` method as we displayed in the previous section:
from conans import ConanFile, CMake, tools

class HelloConan(ConanFile):
    ...

    def source(self):
        self.run("git clone https://github.com/conan-io/hello.git")
    ...

You can also use the tools.Git class:

from conans import ConanFile, CMake, tools

class HelloConan(ConanFile):
    ...

    def source(self):
        git = tools.Git(folder="hello")
        git.clone("https://github.com/conan-io/hello.git", "master")
    ...

2. Using the scm attribute of the ConanFile:

Warning: This is an experimental feature subject to breaking changes in future releases.

from conans import ConanFile, CMake, tools

class HelloConan(ConanFile):
    scm = {
        "type": "git",
        "subfolder": "hello",
        "url": "https://github.com/conan-io/hello.git",
        "revision": "master"
    }

Conan will clone the scm url and will checkout the scm revision. Head to creating package documentation to know more details about SCM feature.

The source() method will be called after the checkout process, so you can still use it to patch something or retrieve more sources, but it is not necessary in most cases.

5.3 Recipe and Sources in the Same Repo

Sometimes it is more convenient to have the recipe and source code together in the same repository. This is true especially if you are developing and packaging your own library, and not one from a third-party.

There are two different approaches:

- Using the exports sources attribute of the conanfile to export the source code together with the recipe. This way the recipe is self-contained and will not need to fetch the code from external origins when building from sources. It can be considered a “snapshot” of the source code.
- Using the scm attribute of the conanfile to capture the remote and commit of your repository automatically.
5.3.1 Exporting the Sources with the Recipe: exports_sources

This could be an appropriate approach if we want the package recipe to live in the same repository as the source code it is packaging.

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

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

A `src` folder will be created with the same “hello” source code as in the previous example. You can have a look at it and see that the code is straightforward.

Now let’s have a look at `conanfile.py`:

```python
from conans import ConanFile, CMake

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

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

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

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

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

There are two important changes:

- Added the `exports_sources` field, indicating to Conan to copy all the files from the local `src` folder into the package recipe.
- Removed the `source()` method, since it is no longer necessary to retrieve external sources.

Also, you can notice the two CMake lines:

```
include(${CMAKE_BINARY_DIR}/conanbuildinfo.cmake)
conan_basic_setup()
```
They are not added in the package recipe, as they can be directly added to the src/CMakeLists.txt file.

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

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

### 5.3.2 Capturing the Remote and Commit: scm

**Warning:** This is an experimental feature subject to breaking changes in future releases.

You can use the `scm` attribute with the `url` and `revision` field set to `auto`. When you export the recipe (or when `conan create` is called) the exported recipe will capture the remote and commit of the local repository:

```python
from conans import ConanFile, CMake, tools

class HelloConan(ConanFile):
    scm = {
        "type": "git",  # Use "type": "svn", if local repo is managed using SVN
        "subfolder": "hello",
        "url": "auto",
        "revision": "auto"
    }
...
```

You can commit and push the `conanfile.py` to your origin repository, which will always preserve the `auto` values. But when the file is exported to the Conan local cache, the copied recipe in the local cache will point to the captured remote and commit:

```python
from conans import ConanFile, CMake, tools

class HelloConan(ConanFile):
    scm = {
        "type": "git",
        "subfolder": "hello",
        "url": "https://github.com/conan-io/hello.git",
        "revision": "437676e15da7090a1368255097f51b1a470905a0"
    }
...
```

So when you upload the recipe to a Conan remote, the recipe will contain the “resolved” URL and commit.

When you are requiring your HelloConan, the `conan install` will retrieve the recipe from the remote. If you are building the package, the source code will be fetched from the captured url/commit.

As SCM attributes are evaluated in the workspace context (see `scm` attribute), you can write more complex functions to retrieve the proper values, this source `conanfile.py` will be valid too:

```python
import os
from conans import ConanFile, CMake, tools

def get_remote_url():
    """ Get remote url regardless of the cloned directory """

(continues on next page)
```
here = os.path.dirname(__file__)
svn = tools.SVN(here)
return svn.get_remote_url()
```

```python
class HelloConan(ConanFile):
    scm = {
        "type": "svn",
        "subfolder": "hello",
        "url": get_remote_url(),
        "revision": "auto"
    }
    ...
```

**Tip:** When doing a `conan create` of a recipe using `scm`, Conan will save the path to the local source repository. Every time the `conan create` command is invoked, the sources will not be downloaded from the remote repository but copied from the local directory.

This allows to build packages making changes to the source code without the need of committing them and pushing them to the remote repository. This convenient to speed up the development of your packages when cloning from a local repository.

**Warning:** This optimization can lead to non-reproducible packages if changes in the source code are not committed and the recipe is uploaded with its packages to a remote.

### 5.4 Packaging Existing 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 are not using Conan. Under these circumstances building from sources is not what you want. You should package the local files in the following situations:

- When you cannot build the packages from sources (when only pre-built binaries are available).
- When you are developing your package locally and you want to export the built artifacts to the local cache.
  As you don’t want to rebuild again (clean copy) your artifacts, you don’t want to call `conan create`. This method will keep your build cache if you are using an IDE or calling locally to the `conan build` command.

#### 5.4.1 Packaging Pre-built Binaries

Running the `build()` method, when the files you want to package are local, results in no added value as the files copied from the user folder cannot be reproduced. For this scenario, run `conan export-pkg` command directly.

A Conan recipe is still required, but is very simple and will only include the package meta information. A basic recipe can be created with the `conan new` command:

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

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

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

(continues on next page)
The provided `package_info()` method scans the package files to provide end-users with the name of the libraries to link to. This method can be further customized to provide additional build flags (typically dependent on the settings). The default `package_info()` applies as follows: it defines headers in the “include” folder, libraries in the “lib” folder, and binaries in the “bin” folder. A different package layout can be defined in the `package_info()` method.

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

Based on the above, We can assume that our current directory contains a `lib` folder with a number binaries for this “hello” library `libhello.a`, compatible for example with Windows MinGW (gcc) version 4.9:

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

Having a `test_package` folder is still highly recommended for testing the package locally before upload. As we don’t want to build the package from the sources, the flow would be:

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

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

### 5.4.2 Downloading and Packaging Pre-built Binaries

In this scenario, 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. Follow our sample recipe for this purpose:

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

    def build(self):
        if self.settings.os == "Windows" and self.compiler == "Visual Studio":
            url = ("https://<someurl>/downloads/hello_binary%s%s.zip" % (str(self.settings.compiler.version), str(self.settings.build_type)))
        elif ...:
            url = ...
        else:
            raise Exception("Binary does not exist for these settings")
        tools.get(url)
```
def package(self):
    self.copy("*") # assume package as-is, but you can also copy specific files, or rearrange

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

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

Note:
• This is a standard Conan package even if the binaries are being retrieved from elsewhere. The recommended approach is to use conan create, and include a small consuming project in addition to the above recipe, to test locally and then proceed to upload the Conan package with the binaries to the 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 packages could be to define a build_policy="missing", especially if the URLs are also under the team control. If they are external (on the internet), it could be better to create the packages and store them on your own Conan server, so that the builds do not rely on third party URL being available.

5.5 Understanding Packaging

5.5.1 Creating and Testing Packages Manually

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

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

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

$ conan create . demo/testing

This is equivalent to:

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

Once the package is created, it can be consumed like any other package, by adding Hello/0.1@demo/testing to a project conanfile.txt or conanfile.py requirements and running:

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

5.5. Understanding Packaging
5.5.2 Package Creation Process

It is very useful for package creators and Conan users in general to understand the flow for creating a package inside the conan local cache, and all about its layout.

Each package recipe contains five important folders in the **local cache**:

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

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

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

The `export` and `export_source` files are copied to the `source` folder, and then the `source()` method is executed (if
it exists). Note that there is only one source folder for all the binary packages. If when generating the code, there is source code that varies for the different configurations, it cannot be generated using the source() method, but rather needs to be generated using the build() method.

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

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

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

Any doubts? Please check out our FAQ section or .

5.6 Defining Package ABI Compatibility

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

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

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

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

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

The process is:

1. Conan gets the user input settings and options. Those settings and options can come from the command line, profiles or from the values cached in the latest conan install execution.

2. Conan retrieves the MyLib/1.0@user/channel recipe, reads the settings attribute, and assigns the necessary values.

3. With the current package values for settings (also options and requires), it will compute a SHA1 hash that will serve as the binary package ID, e.g., c6d75a933080ca17eb7f076813e7fb21aaa740f2.

4. Conan will try to find the c6d75... binary package. If it exists, it will be retrieved. If it cannot be found, it will fail and indicate that it can be built from sources using conan install --build.

If the package is installed again using different settings, for example, on a 32-bit architecture:

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

The process will be repeated with a different generated package ID, because the arch setting will have a different value. The same applies to different compilers, compiler versions, build types. When generating multiple binaries - a separate ID is generated for each configuration.

When developers using the package use the same settings as one of those uploaded binaries, the computed package ID will be identical causing the binary to be retrieved and reused without the need of rebuilding it from the sources.
The `options` behavior is very similar. The main difference is that `options` can be more easily defined at the package level and they can be defaulted. Check the `options` reference.

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

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

No matter the settings are defined by the users, including the compiler or version, the package settings and options will always be the same (left empty) and they will hash to the same binary package ID. That package will typically contain just the header files.

What happens if we have a library that we can be built with GCC 4.8 and will preserve the ABI compatibility with GCC 4.9? (This kind of compatibility is easier to achieve for example for pure C libraries).

Although it could be argued that it is worth rebuilding with 4.9 too -to get fixes and performance improvements-. Let’s suppose that we don’t want to create 2 different binaries, but just a single built with GCC 4.8 which also needs to be compatible for GCC 4.9 installations.

### 5.6.1 Defining a Custom `package_id()`

The default `package_id()` uses the `settings` and `options` directly as defined, and assumes the semantic versioning for dependencies is defined in `requires`.

This `package_id()` method can be overridden to control the package ID generation. Within the `package_id()`, we have access to the `self.info` object, which is hashed to compute the binary ID and contains:

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

Initially this `info` object contains the original settings and options, but they can be changed without constraints to any other string value.

For example, if you are sure your package ABI compatibility is fine for GCC versions > 4.5 and < 5.0, you could do the following:

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

class PkgConan(ConanFile):
    name = "Pkg"
    version = "1.0"
    settings = "compiler", "build_type"
    def package_id(self):
        v = Version(str(self.settings.compiler.version))
        if self.settings.compiler == "gcc" and (v >= "4.5" and v < "5.0"):
            self.info.settings.compiler.version = "GCC version between 4.5 and 5.0"
```

We have set the `self.info.settings.compiler.version` with an arbitrary string, the value of which is not important (could be any string). The only important thing is that it is the same for any GCC version between 4.5 and 5.0. For all those versions, the compiler version will always be hashed to the same ID.
Let’s try and check that it works properly when installing the package for GCC 4.5:

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

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

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

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

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

The required package has the same result again af04...46ad. Now we can try using GCC 4.4 (< 4.5):

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

Requirements
Pkg/1.0@myuser/mychannel from local
Packages
Pkg/1.0@myuser/mychannel:7d02dc01581029782b59dccc8c9783a73ab3c22dd

The computed package ID is different which means that we need a different binary package for GCC 4.4.

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

**See also:** Check `package_id()` to see the available helper methods and change its behavior for things like:

- Recipes packaging **header only** libraries.
- Adjusting **Visual Studio toolsets** compatibility.

### 5.6.2 Dependency Issues

Let’s define a simple scenario whereby there are two packages: MyOtherLib/2.0 and MyLib/1.0 which depends on MyOtherLib/2.0. Let’s assume that their recipes and binaries have already been created and uploaded to a Conan remote.

Now, a new release for MyOtherLib/2.1 is released with an improved recipe and new binaries. The MyLib/1.0 is modified and is required to be upgraded to MyOtherLib/2.1.

**Note:** This scenario will be the same in the case that a consuming project of MyLib/1.0 defines a dependency to MyOtherLib/2.1, which takes precedence over the existing project in MyLib/1.0.

The question is: **Is it necessary to build new MyLib/1.0 binary packages?** or are the existing packages still valid? The answer: **It depends.**
Let’s assume that both packages are compiled as static libraries and that the API exposed by `MyOtherLib` to `MyLib/1.0` through the public headers, has not changed at all. In this case, it is not required to build new binaries for `MyLib/1.0` because the final consumer will link against both `Mylib/1.0` and `MyOtherLib/2.1`.

On the other hand, it could happen that the API exposed by `MyOtherLib` in the public headers has changed, but without affecting the `MyLib/1.0` binary for any reason (like changes consisting on new functions not used by `MyLib`). The same reasoning would apply if `MyOtherLib` was only the header.

But what if a header file of `MyOtherLib` -named `myadd.h`- has changed from 2.0 to 2.1:

<table>
<thead>
<tr>
<th>Listing 1: <code>myadd.h</code> header file in version 2.0</th>
</tr>
</thead>
</table>
| ```
| int addition (int a, int b) { return a - b; }
| ```
<p>| |
|  |</p>
<table>
<thead>
<tr>
<th>Listing 2: <code>myadd.h</code> header file in version 2.1</th>
</tr>
</thead>
</table>
| ```
| int addition (int a, int b) { return a + b; }
| ```

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

Then, a new binary for `MyLib/1.0` is required to be built for the new dependency version. Otherwise it will maintain the old, buggy `addition()` version. Even in the case that `MyLib/1.0` doesn’t have any change in its code lines neither in the recipe, the resulting binary rebuilding `MyLib` requires `MyOtherLib/2.1` and the package to be different.

### 5.6.3 Using package_id() for Package Dependencies

The `self.info` object has also a `requires` object. It is a dictionary containing the necessary information for each requirement, all direct and transitive dependencies. For example, `self.info.requires["MyOtherLib"]` is a `RequirementInfo` object.

- Each `RequirementInfo` has the following read only reference fields:
  - full_name: Full require’s name, e.g., `MyOtherLib`
  - full_version: Full require’s version, e.g., `1.2`
  - full_user: Full require’s user, e.g., `my_user`
  - full_channel: Full require’s channel, e.g., `stable`
  - full_package_id: Full require’s package ID, e.g., `c6d75a...`

- The following fields are used in the `package_id()` evaluation:
  - name: By default same value as full_name, e.g., `MyOtherLib`
  - version: By default the major version representation of the full_version. E.g., `1.Y` for a `1.2` `full_version` field and `1.Y.Z` for a `1.2.3` `full_version` field.
  - user: By default None (doesn’t affect the package ID).
  - channel: By default None (doesn’t affect the package ID).
  - package_id: By default None (doesn’t affect the package ID).

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

- The versioning schema followed by our requirements (semver?, custom?).
- The type of library being built or reused (shared (`.so`, `.dll`, `.dylib`), static).
Versioning Schema

By default Conan assumes semver compatibility. For example, if a version changes from minor 2.0 to 2.1, Conan will assume that the API is compatible (headers not changing), and that it is not necessary to build a new binary for it. This also applies to patches, whereby changing from 2.1.10 to 2.1.11 doesn’t require a re-build.

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

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

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

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

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

        # Changes in major and minor versions will change the Package ID but
        # only a MyOtherLib patch won't. E.g., from 1.2.3 to 1.2.89 won't change.
        myotherlib.version = myotherlib.full_version.minor()
```

Besides version, there are additional helpers that can be used to determine whether the channel and user of one dependency also affects the binary package, or even the required package ID can change your own package ID.

You can determine if the following variables within any requirement change the ID of your binary package using the following modes:

<table>
<thead>
<tr>
<th>Modes / Variables</th>
<th>name</th>
<th>version</th>
<th>user</th>
<th>channel</th>
<th>package_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>semver_direct_mode()</td>
<td>Yes</td>
<td>Yes, only &gt; 1.0.0 (e.g., 1.2.Z+b102)</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>semver_mode()</td>
<td>Yes</td>
<td>Yes, only &gt; 1.0.0 (e.g., 1.2.Z+b102)</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>major_mode()</td>
<td>Yes</td>
<td>Yes (e.g., 1.2.Z+b102)</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>minor_mode()</td>
<td>Yes</td>
<td>Yes (e.g., 1.2.Z+b102)</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>patch_mode()</td>
<td>Yes</td>
<td>Yes (e.g., 1.2.3+b102)</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>base_mode()</td>
<td>Yes</td>
<td>Yes (e.g., 1.7+b102)</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>full_version_mode()</td>
<td>Yes</td>
<td>Yes (e.g., 1.2.3+b102)</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>full_recipe_mode()</td>
<td>Yes</td>
<td>Yes (e.g., 1.2.3+b102)</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>full_package_mode()</td>
<td>Yes</td>
<td>Yes (e.g., 1.2.3+b102)</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>unrelated_mode()</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

All the modes can be applied to all dependencies, or to individual ones:

```python
def package_id(self):
    # apply semver_mode for all the dependencies of the package
    self.info.requires.semver_mode()

    # use semver_mode just for MyOtherLib
    self.info.requires["MyOtherLib"].semver_mode()
```
• **semver_direct_mode()**: This is the default mode. It uses `semver_mode()` for direct dependencies (first level dependencies, directly declared by the package) and `unrelated_mode()` for indirect, transitive dependencies of the package. It assumes that the binary will be affected by the direct dependencies, which they will already encode how their transitive dependencies affect them. This might not always be true, as explained above, and that is the reason it is possible to customize it.

In this mode, if the package depends on “MyLib”, which transitively depends on “MyOtherLib”, the mode means:

| MyLib/1.2.3@user/testing | => MyLib/1.Y.Z |
| MyOtherLib/2.3.4@user/testing | => |

So the direct dependencies are mapped to the major version only. Changing its channel, or using version `MyLib/1.4.5` will still produce `MyLib/1.Y.Z` and thus the same package-id. The indirect, transitive dependency doesn’t affect the package-id at all.

• **semver_mode()**: In this mode, only a major release version (starting from **1.0.0**) changes the package ID. Every version change prior to 1.0.0 changes the package ID, but only major changes after 1.0.0 will be applied.

```python
def package_id(self):
    self.info.requires["MyOtherLib"].semver_mode()
```

This results in:

| MyLib/1.2.3@user/testing | => MyLib/1.Y.Z |
| MyOtherLib/2.3.4@user/testing | => MyOtherLib/2.Y.Z |

In this mode, versions starting with **0** are considered unstable and mapped to the full version:

| MyLib/0.2.3@user/testing | => MyLib/0.2.3 |
| MyOtherLib/0.3.4@user/testing | => MyOtherLib/0.3.4 |

• **major_mode()**: Any change in the major release version (starting from **0.0.0**) changes the package ID.

```python
def package_id(self):
    self.info.requires["MyOtherLib"].major_mode()
```

This mode is basically the same as `semver_mode`, but the only difference is that major versions `0.Y.Z`, which are considered unstable by semver, are still mapped to only the major, dropping the minor and patch parts.

• **minor_mode()**: Any change in major or minor (not patch nor build) version of the required dependency changes the package ID.

```python
def package_id(self):
    self.info.requires["MyOtherLib"].minor_mode()
```

• **patch_mode()**: Any changes to major, minor or patch (not build) versions of the required dependency change the package ID.

```python
def package_id(self):
    self.info.requires["MyOtherLib"].patch_mode()
```

• **base_mode()**: Any changes to the base of the version (not build) of the required dependency changes the package ID. Note that in the case of semver notation this may produce the same result as `patch_mode()`, but it is actually intended to dismiss the build part of the version even without strict semver.

```python
def package_id(self):
    self.info.requires["MyOtherLib"].base_mode()
```
• **full_version_mode()**: Any changes to the version of the required dependency changes the package ID.

```python
def package_id(self):
    self.info.requires["MyOtherLib"].full_version_mode()
```

MyOtherLib/1.3.4-a4+b3@user/testing => MyOtherLib/1.3.4-a4+b3

• **full_recipe_mode()**: Any change in the reference of the requirement (user & channel too) changes the package ID.

```python
def package_id(self):
    self.info.requires["MyOtherLib"].full_recipe_mode()
```

This keeps the whole dependency reference, except the package-id of the dependency.

MyOtherLib/1.3.4-a4+b3@user/testing => MyOtherLib/1.3.4-a4+b3@user/testing

• **full_package_mode()**: Any change in the required version, user, channel or package ID changes the package ID.

```python
def package_id(self):
    self.info.requires["MyOtherLib"].full_package_mode()
```

This is the stricter mode. Any change to the dependency, including its binary package-id, will in turn produce a new package-id for the consumer package.

MyOtherLib/1.3.4-a4+b3@user/testing:73b..fa56 => MyOtherLib/1.3.4-a4+b3@user/testing:73b..fa56

• **unrelated_mode()**: Requirements do not change the package ID.

```python
def package_id(self):
    self.info.requires["MyOtherLib"].unrelated_mode()
```

You can also adjust the individual properties manually:

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

    # Same as myotherlib.semver_mode()
    myotherlib.name = myotherlib.full_name
    myotherlib.version = myotherlib.full_version.stable()  # major(), minor(),
    # patch(), base, build
    myotherlib.user = myotherlib.channel = myotherlib.package_id = None

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

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

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

5.6. Defining Package ABI Compatibility 49
Changing the default package-id mode

It is possible to change the default `semver_direct_mode` package-id mode, in the `conan.conf` file:

```
[general]
default_package_id_mode=full_package_mode
```

Possible values are the names of the above methods: `full_recipe_mode`, `semver_mode`, etc.

Note that the default package-id mode is the mode that is used when the package is initialized and before `package_id()` method is called. You can still define `full_package_mode` as default in `conan.conf`, but if a recipe declare that it is header-only, with:

```python
def package_id(self):
    self.info.header_only()  # clears requires, but also settings if existing
    # or if there are no settings/options, this would be equivalent
    self.info.requires.clear()  # or self.info.requires.unrelated_mode()
```

That would still be executed, changing the “default” behavior, and leading to a package that only generates 1 package-id for all possible configurations and versions of dependencies.

Remember that `conan.conf` can be shared and installed with `conan config install`.

Library Types: Shared, Static, Header-only

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

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

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

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

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

  It depends. If the public headers have not changed at all, it is not necessary. Actually it might be necessary to consider transitive dependencies that are shared among the public headers, how they are linked and if they cross the frontiers of the API, it might also lead to incompatibilities. If the public headers have changed, it would depend on what changes and how are they used in `MyLib/1.0`. Adding new methods to the public headers will have no impact, but changing the implementation of some functions that will be inlined when compiled from `MyLib/1.0` will definitely require re-building. For this case, it could make sense to have this configuration:
def package_id(self):
    # Any change in the MyOtherLib version, user or channel
    # or Package ID will affect our package ID
    self.info.requires["MyOtherLib"].full_package_mode()

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

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

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

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

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

```python
def package_id(self):
    self.info.requires["MyOtherLib"].full_package()
    self.info.requires["MyOtherLib"].channel = None  # Channel doesn't change out
    →package ID
```

### 5.7 Inspecting Packages

You can inspect the uploaded packages and also the packages in the local cache by running the `conan get` command.

• List the files of a local recipe folder:

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

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

• Print the `conaninfo.txt` file of a binary package:

```bash
$ conan get zlib/1.2.11@conan/stable -p 2144f833c251030c3cfd61c4354ae0e38607a909
```

• Print the `conanfile.py` from a remote package:

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

```python
from conans import ConanFile, tools, CMake, AutoToolsBuildEnvironment
from conans.util import import files
```

(continues on next page)
from conans import __version__ as conan_version
import os

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

#...

Check the conan get command command reference and more examples.

5.8 Packaging Approaches

Package recipes have three methods for controlling the package’s binary compatibility and for implementing different packaging approaches: package_id(), build_id() and package_info().

These methods let package creators select the method most suitable for each library.

5.8.1 1 config (1 build) -> 1 package

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

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

Using this approach, the package_info() method, allows you to set the appropriate values for consumers, letting them know about the package library names, necessary definitions and compile flags.
It is very important to note that it is declaring the `build_type` as a setting. This means that a different package will be generated for each different value of such setting.

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

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

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

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

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

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

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

Read more about this in `package_info()`.

### 5.8.2 N configs -> 1 package

You may want to package both debug and release artifacts in the same package, so it can be consumed from IDEs like Visual Studio. This will change the debug/release configuration from the IDE, without having to specify it in the command line. This type of package can contain different artifacts for different configurations, and can be used for example to include both the release and debug version of the “hello” library, in the same package.
A complete working example of the following code can be found in a github repo. You should be able to run:

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

Creating a multi-configuration Debug/Release package is simple, see the following example using CMake.

The first step will be to remove `build_type` from the settings. It will not be an input setting, the generated package will always be the same, containing both Debug and Release artifacts. The Visual Studio runtime is different for debug and release (MDd or MD) and is set using the default runtime (MD/MDd). If this meets your needs, we recommend removing the runtime subsetting in the `configure()` method:

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

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

    def build(self):
        cmake = CMake(self)
        if cmake.is_multi_configuration:
            cmd = "cmake "$s" "$s" % (self.source_folder, cmake.command_line)
            self.run(cmd)
            self.run("cmake --build . --config Debug")
            self.run("cmake --build . --config Release")
        else:
            for config in ("Debug", "Release"):
                self.output.info("Building $s" % config)
                self.run("cmake "$s" -DCMAKE_BUILD_TYPE=$s\
                         % (self.source_folder, cmake.command_line, config))
```

(continues on next page)
In this case, we assume that the binaries will be differentiated with a suffix in the Cmake syntax:

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

This will translate to the cmake variables:

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

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

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

**Important:** The above code assumes that the package will always use the default Visual Studio runtime (MD/MDd). To keep the package configurable for supporting static(MT)/dynamic(MD) linking with the VS runtime library, do the following:

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

All these steps are already coded in the repo [https://github.com/memsharded/hello_multi_config](https://github.com/memsharded/hello_multi_config) and commented out as “Alternative 2”

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

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

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

5.8. Packaging Approaches 55
Note: The automatic conversion of multi-config variables to generators is currently only implemented in the cmake and txt generators. If you want to have support for them in another build system, please open a GitHub issue.

5.8.3 N configs (1 build) -> N packages

It’s possible that an existing build script is simultaneously building binaries for different configurations, like debug/release, or different architectures (32/64bits), or library types (shared/static). If such a build script is used in the previous “Single configuration packages” approach, it will definitely work without problems. However, we’ll be wasting precious build time, as we’ll be re-building the rebuilding project for each package, then extracting the relevant artifacts for the relevant configuration, while ignoring the others.

It is more efficient to build the logic, whereby the same build can be reused to create different packages:

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

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

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

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

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

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

Read more about this in `build_id()`.

## 5.9 Package Creator Tools

Using Python (or just pure shell or bash) scripting, allows you to easily automate the whole package creation and testing process, for many different configurations. For example you could put the following script in the package root folder. Name it `build.py`:

```python
import os, sys
import platform

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

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

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

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

```
$ python build.py
```

We have developed another FOSS tool for package creators, the **Conan Package Tools** to help you generate multiple binary packages from a package recipe. It offers a simple way to define the different configurations and to call `conan test`. In addition to offering CI integration like **Travis CI**, Appveyor and Bamboo, for cloud-based automated binary package creation, testing, and uploading.

This tool enables the creation of hundreds of binary packages in the cloud with a simple `git push` and supports:

- **Easy generation of multiple Conan packages** with different configurations.
- Automated/remote package generation in **Travis/Appveyor** server with distributed builds in CI jobs for big/slow builds.
- **Docker**: Automatic generation of packages for several versions of **gcc** and **clang** in Linux, and in Travis CI.
- **Visual Studio**: Automatic configuration of the command line environment with detected settings.

It’s available in pypi:

```
$ pip install conan_package_tools
```

For more information, read the README.md in the **Conan Package Tools** repository.
This section shows how to upload packages using remotes and specifies the different binary repositories you can use.

### 6.1 Remotes

In the previous sections, we built several packages on our computer that were stored in the local cache, typically under `~/.conan/data`. Now, you might want to upload them to a Conan server for later use on another machine, project, or for sharing purposes.

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

There are several possibilities when uploading packages to a server:

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.

- **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](#).

- **Conan server**: Simple, free and open source, MIT licensed server that comes bundled with the Conan client. Check [Running conan_server](#) for more information.

For distribution:

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

#### 6.1.1 Bintray Official Repositories

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

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

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

Conan Transit

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

Deprecated. Contains mostly outdated packages some of which are not compatible with the latest Conan versions, so refrain from using them. This repository only exists for backward compatibility purposes. It is not a default remote in the Conan client and will be completely removed soon. This repository is an exact duplicate of the old server.conan.io repository at June 11, 2017 08:00 CET. It’s a read-only repository, allowing you to only download hosted packages.

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

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

6.1.2 Bintray Community Repositories

There are a number of popular community repositories that may be of interest for Conan users for retrieving open source packages. A number of these repositories are not affiliated with the Conan team.

Bincrafters

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

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

Use the following command to add this remote to Conan:

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

Conan Community

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

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

Use the following command to add this remote to Conan:

```
$ conan remote add conan-community https://api.bintray.com/conan/conan-community/conan
```
Note: If you are working in a team, you probably want to use the same remotes everywhere: developer machines, CI. The `conan config install` command can automatically define the remotes in a Conan client, as well as other resources as profiles. Have a look at the `conan config install` command.

### 6.2 Uploading Packages to Remotes

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

```
$ conan remote list
```

You can easily add any remote. To run a remote on your machine:

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

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

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

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

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

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

The `--all` option will upload the package recipe plus all the binary packages. Omitting the `--all` option will upload the package recipe only. For fine-grained control over which binary packages are upload to the server, consider using the `--packages` or `--query` flags. `--packages` allows you to explicitly declare which package gets uploaded to the server by specifying the package ID. `--query` accepts a query parameter, e.g. `arch=armv8` and `os=Linux`, and only uploads binary packages which match this query. When using the `--query` flag, ensure that your query string is enclosed in quotes to make the parameter explicit to your shell. For example, `conan upload <package> -q 'arch=x86_64 and os=Linux' ...` is appropriate use of the `--query` flag.

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/0.1@demo/testing -r=my_local_server
```

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

Now we can check if we 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*
$ conan search
```

### 6.2. Uploading Packages to Remotes

61
Since we have our test setup from the previous section, we can just use it for our test. Go to your package folder and run the tests again, now saying that we don’t want to build the sources again. We just want to check if we can download the binaries and use them:

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

You will see that the test is built, but the packages are not. The binaries are simply downloaded from your local server.

You can check their existence on your local computer again with:

```bash
$ conan search
```

### 6.3 Using Bintray

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

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

#### 6.3.1 Uploading to Bintray

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

1. **Create a Bintray Open Source account**
   
   Browse to [https://bintray.com/signup/oss](https://bintray.com/signup/oss) and submit the form to create your account. Note that you don’t have to use the same username that you use for your Conan account.

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

2. **Create a Conan repository**
   
   If you intend to collaborate with other users, you first need to create a Bintray organization, and create your repository under the organization’s profile rather than under your own user profile.
   
   In your user profile (or organization profile), click “Add new repository” and fill in the Create Repository form. Make sure to select Conan as the Type.

3. **Add your Bintray repository**
   
   Add a Conan remote in your Conan client pointing to your Bintray repository:
   
   ```bash
   $ conan remote add <REMOTE> <YOUR_BINTRAY_REPO_URL>
   ```
   
   Use the Set Me Up button on your repository page on Bintray to get its URL.

4. **Get your API key**
   
   Your API key is the “password” used to authenticate the Conan client to Bintray, NOT your Bintray password. To get your API key, go to “Edit Your Profile” in your Bintray account and check the API Key section.
5. Set your user credentials

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

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

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

1. conan-center
2. Your own repository

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

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

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

### 6.3.2 Contributing Packages to Conan-Center

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

Your request is dealt with differently depending on the submitted package type:

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

- **If you are packaging a third-party library,** follow these guidelines:

Contributing a library to Conan-Center is really straightforward when you know how to *upload your packages to your own Bintray repository*. All you have to do is to navigate to the main page of the package in Bintray and click the “Add to Conan Center” button to start the inclusion request process.
Inclusion Guidelines for Third-Party libraries

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

A Single Conan Package Per OSS Library

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

• Ensure that there is no additional Conan package for the same library. If you are planning to support a new version of a library that already exists in the conan-center repository, please contact the package author and collaborate. All the versions of the same library have to be on the same Bintray Conan package.

• It is recommended to contact the library author and suggest to maintain the Conan package. When possible, open a pull request to the original repository of the library with the required Conan files or suggest to open a new repository with the recipe.

• If you are going to collaborate with different users to maintain the Conan package, open a Bintray organization.

Recipe Quality

• Git public repository: The recipe needs to be hosted in a public Git repository that supports collaboration.

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

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

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

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

Sometimes there is no `license` file, for that, read `Howtos/Extract licenses from headers`.
- **Invalid configurations**: There is a special exception, `conans.errors.ConanInvalidConfiguration` to be launched from `configure()` function in a recipe if the given configuration/options is known not to work. This way the recipe owner can declare an invalid configuration and consumers (e.g. CI tools like `conan-package-tools`) will be able to handle it.

### CI Integration

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

You can also copy the following files from this zlib Conan package repository and modify them:
  – `.travis` folder. No need to adjust anything.
  – `.travis.yml` file. Adjust your username, library reference, etc.
  – `appveyor.yml` file. Adjust your username, library reference, etc.

• Take a look at the Travis CI, Appveyor and GitLab CI integration guides.

**Bintray Package Data**

In the Bintray page of your package, fill in the following fields:

• Description (description of the packaged library)

• Licenses (license of the packaged library)

• Tags

• Maturity

• Website: If any, website of the library

• Issues tracker: URL of the issue tracker from your github repository e.g., https://github.com/conan-community/community/issues

• Version control: URL of your recipe github repository, e.g., https://github.com/conan-community/conan-zlib

• GitHub repo (user/repo): e.g., conan-community/conan-zlib

For each version page (optional, but recommended):

• Select the README from github.

• Select the Release Notes.

### 6.4 Artifactory Community Edition for C/C++

Artifactory Community Edition (CE) for C/C++ is the recommended server for development and hosting private packages for a team or company. It is completely free, and it features a WebUI, 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.

#### 6.4.1 Running Artifactory CE

There are several ways to download and run Artifactory CE. The simplest one might be to download and unzip the designated zip file, though other installers, including also installing from a Docker image. When the file is unzipped, launch Artifactory by double clicking the `.bat` or `.sh` script in the `bin` subfolder, depending on the OS. Java 8 update 45 or later runtime is required. If you don’t have it, please install it first (newer Java versions preferred).
Once Artifactory has started, navigate to the default URL http://localhost:8081, where the Web UI should be running. The default user and password are admin:password.

### 6.4.2 Creating and Using a Conan Repo

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

Now, it is necessary to configure the client. Go to Artifacts, and click on the created repository. The “Set Me Up” button in the top right corner provides instructions on how to configure the remote in the Conan client:

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

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

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

### 6.4.3 Migrating from Other Servers

If you are already running another server, for example, the open source `conan_server`, it is easy to migrate your packages, using the Conan client to download the packages and re-upload them to the new server.

This Python script might be helpful, given that it already defines the respective local and artifactory remotes:

```python
import os
import subprocess

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

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

for package in packages:
    print("Downloading %s" % package)
    run("conan download %s -r=local" % package)
run("conan upload * --all --confirm -r=artifactory")

6.5 Running conan_server

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

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

```
$ conan_server
```

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

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

6.5.1 Running from Source (linux)

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

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

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

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

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

Note: Please note the timeout of -t 300 seconds, resulting in a 5 minute parameter. If your transfers are very large or on a slow network, you might need to increase that value.
You can also bind to an IPv6 address or specify both IPv4 and IPv6 addresses:

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

### 6.5.2 Server Configuration

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

The server configuration file is by default:

```ini
[server]
jwt_secret: MnpuzsExftskYGOMgaTYDKfw
jwt_expire_minutes: 120
ssl_enabled: False
port: 9300
public_port:
host_name: localhost
store_adapter: disk
authorize_timeout: 1800

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

updown_secret: NyiSWNWnwumTVpGpoANuyyhR

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

[read_permissions]
# opencv/1.2.3@lasote/testing: default_user default_user2
# By default all users can read all blocks
*/*/*/*: *

[users]
demo: demo
```

**Server Parameters**

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

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

- **host_name**: If you set `host_name`, you must use the machine’s IP where you are running your server (or domain name), something like `host_name: 192.168.1.100`. This IP (or domain name) has to be visible (and resolved) by the Conan client, so take it 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


To learn more, see How to manage SSL (TLS) certificates.

Conan has implemented an extensible storage backend based on the abstract class StorageAdapter. Currently, the server only supports storage on disk. 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.

Running the Conan Server with SSL using Nginx

```
server.conf

[server]
port: 9300
```

nginx conf file

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

(continues on next page)
remote configuration in Conan client

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

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/

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/python2.7/dist-packages/conans/server/server_launcher.py
  WSGICallableObject app
  WSGIPassAuthorization On
  <Directory /usr/local/lib/python2.7/dist-packages/conans>
    Require all granted
  </Directory>
</VirtualHost>
If you want to use Conan checked out from source in, for example in /srv/conan, the conf file should be as follows:

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

```xml
<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.

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

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

DEVELOPING PACKAGES

This section shows how to work on packages with source code continuously being modified.

7.1 Package development flow

In the previous examples, we used the `conan create` command to create a package of our library. Every time it is run, Conan performs the following costly operations:

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

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

The following section describes the local development flow, based on the Bincrafters community blog.

The 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.

We will use this conan flow example to follow the steps in the order below.

7.1.1 conan source

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

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

<table>
<thead>
<tr>
<th>Input folders</th>
<th>Output folders</th>
</tr>
</thead>
<tbody>
<tr>
<td>–</td>
<td>source-folder</td>
</tr>
</tbody>
</table>
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.

### 7.1.2 conan install

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

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

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

### 7.1.3 conan build

The build method takes a path to a folder that has sources and also to the install folder to get the information of the settings and dependencies. It uses a path to a folder where it will perform the build. In this case, as we are including the `conanbuildinfo.cmake` file, we will use the folder from the install step.

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

Here we can avoid the repetition of `--install-folder=tmp/build` and it will be defaulted to the `--build-folder` value.
This is pretty straightforward, but it does add a very helpful new shortcut for people who are packaging their own library. Now, developers can make changes in their normal source directory and just pass that path as the --source-folder.

### 7.1.4 conan package

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

<table>
<thead>
<tr>
<th>Input folders</th>
<th>Output folders</th>
</tr>
</thead>
<tbody>
<tr>
<td>source-folder</td>
<td>package-folder</td>
</tr>
<tr>
<td>install-folder</td>
<td></td>
</tr>
<tr>
<td>build-folder</td>
<td></td>
</tr>
</tbody>
</table>

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

PROJECT: Generating the package
PROJECT: Package folder C:\Users\conan\example_conan_flow\tmp\package
PROJECT: Calling package()
PROJECT package(): Copied 1 '.h' files: hello.h
PROJECT package(): Copied 2 '.lib' files: greet.lib, hello.lib
PROJECT: Package 'package' created

### 7.1.5 conan export-pkg

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

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

<table>
<thead>
<tr>
<th>Input folders</th>
<th>Output folders</th>
</tr>
</thead>
<tbody>
<tr>
<td>source-folder</td>
<td></td>
</tr>
<tr>
<td>install-folder</td>
<td></td>
</tr>
<tr>
<td>build-folder</td>
<td></td>
</tr>
<tr>
<td>package-folder</td>
<td></td>
</tr>
</tbody>
</table>

There are 2 modes of operation:

- Using `source-folder` and `build-folder` will use the `package()` method to extract the artifacts from those folders and create the package, directly in the Conan local cache. Strictly speaking, it doesn’t require executing a `conan package` before, as it packages directly from these source and build folders, though `conan package` is still recommended in the dev-flow to debug the `package()` method.

- Using the `package-folder` argument (incompatible with the above 2), will not use the `package()` method, it will create an exact copy of the provided folder. It assumes the package has already been created by a previous `conan package` command or with a `conan build` command with a `build()` method running a `cmake.install()`.
$ conan export-pkg . user/channel --source-folder=tmp/source --build-folder=tmp/build_˓→--profile=myprofile

Packaging to 6cc50b139b9c3d27b3e9042d5f5372d327b3a9f7
Hello/1.1@user/channel: Generating the package
Hello/1.1@user/channel: Package folder C:\Users\conan\conan\data\Hello\1.˓→1\user\channel\package\6cc50b139b9c3d27b3e9042d5f5372d327b3a9f7
Hello/1.1@user/channel: Calling package()
Hello/1.1@user/channel: package(): Copied 2 '.lib' files: greet.lib, hello.lib
Hello/1.1@user/channel: package(): Copied 2 '.lib' files: greet.lib, hello.lib
Hello/1.1@user/channel: Package '6cc50b139b9c3d27b3e9042d5f5372d327b3a9f7' created

### 7.1.6 conan test

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

```bash
$ conan test test_package Hello/1.1@user/channel
Hello/1.1@user/channel (test package): Installing C:\Users\conan\repos\example_conan_˓→flow\test_package\conanfile.py
Requirements
  Hello/1.1@user/channel from local
Packages
  Hello/1.1@user/channel:6cc50b139b9c3d27b3e9042d5f5372d327b3a9f7
Hello/1.1@user/channel: Already installed!
Hello/1.1@user/channel (test package): Generator cmake created conanbuildinfo.cmake
Hello/1.1@user/channel (test package): Generator txt created conanbuildinfo.txt
Hello/1.1@user/channel (test package): Generated conaninfo.txt
Hello/1.1@user/channel (test package): Running build()
...
```

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

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

```bash
$ git clone git@github.com:memsharded/example_conan_flow.git
$ cd example_conan_flow
$ conan source .
$ conan install . -pr=default
$ conan build .
$ conan package .
# So far, this is local. Now put the local binaries in cache
$ conan export-pkg . Hello/1.1@user/testing -pr=default
# And test it, to check it is working in the local cache
$ conan test test_package Hello/1.1@user/testing
...
Hello/1.1@user/testing (test package): Running test()
Hello World!
```

### 7.1.7 conan create

Now we know we have all the steps of a recipe working. Thus, now is an appropriate time to try to run the recipe all the way through, and put it completely in the local cache.
The usual command for this is `conan create` and it basically performs the previous commands with `conan test` for the `test_package` folder:

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

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

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

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

If you see that the library is also built correctly, you can also skip the `build()` step with the `--keep-build` flag:

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

## 7.2 Packages in editable mode

**Warning:** This is an experimental feature subject to breaking changes in future releases.

When working in big projects with several functionalities interconnected it is recommended to avoid the one-and-only huge project approach in favor of several libraries each one of them specialized in a set of common tasks, even maintained by dedicated teams. This approach helps to isolate and reuse code, helps with compiling times and reduces the likelihood of including files that do not correspond to the API of the required library.

Nevertheless, in some case, it is useful to work in several libraries at the same time and see how the changes in one of them are propagated to the others. Following the local workflow an user can execute the commands `conan source`.

### 7.2. Packages in editable mode

79
**conan install**, **conan build** and **conan package**, but in order to get the changes ready for a consumer library, it is needed the **conan create** that will actually trigger a build to generate the binaries in the cache or to run **conan export-pkg** to copy locally built artifacts into the conan cache and make them available to consumers.

What about if you just can tell Conan where to find the headers and the artifacts ready for consumption in your local working directory? No need to package, just tell Conan to use those artifacts you have just generated with your IDE, sounds good? This is what the feature **editable packages** will do for you.

Let’s see this feature over an example where a developer is creating a **CoolApp** but at the same time they wants to work on cool/version@user/dev library which is tightly coupled to the app.

The package cool/version@user/dev is already working, the developer has the sources in a local folder, they is using whatever method to build and develop locally and can perform a **conan create cool/version@user/dev** to create the package.

Also, there is a **conanfile.txt** (or a more complex recipe) for the application CoolApp that has cool/version@user/dev among its requirements. When building this application, the resources of cool are used from the Conan local cache.

### 7.2.1 Put a package in editable mode

To avoid creating the package cool/version@user/dev 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:

```bash
$ conan editable add <path/to/local/dev/libcool> cool/version@user/dev
```

That is it. Now, every usage of cool/version@user/dev, by any other Conan package, or project will be redirected to the <path/to/local/dev/libcool> user folder, instead of using the package from the conan cache.

Conan package recipes define a package “layout” in their **package_info()** methods. The default one, if nothing is specified is equivalent to:

```python
def package_info(self):
    # default behavior, doesn't need to be explicitly defined in recipes
    self.cpp_info.includedirs = ["include"]
    self.cpp_info.libdirs = ["lib"]
    self.cpp_info.bindirs = ["bin"]
    self.cpp_info.resdirs = ["res"]
```

That means that conan will use the path path/to/local/dev/libcool/include for locating the headers of the cool package, the path/to/local/dev/libcool/lib to locate the libraries of the package, and so on.

That might not be very useful, as typically while editing the source code and doing incremental builds, the development layout is different from that final “package” layout. While it is possible to run a **conan package** local command to execute the packaging in the user folder, and that will achieve that final layout, that is not very elegant. Conan provides several ways to customize the layout for editable packages.

### 7.2.2 Editable packages layouts

The custom layout of a package while it is in editable mode can be defined in different ways:
Recipe defined layout

A recipe can define a custom layout when it is not living in the local cache, in its `package_info()` method, something like:

```python
global import ConanFile

class Pkg(ConanFile):
    settings = "build_type"
    def package_info(self):
        if not self.in_local_cache:
            d = "include_%s" % self.settings.build_type
            self.cpp_info.includedirs = [d.lower()]
```

That will map the include directories to `path/to/local/dev/libcool/include_debug` when working with `build_type=Debug` conan setting, and to `path/to/local/dev/libcool/include_release` if `build_type=Release`. In the same way, other directories (libdirs, bindirs, etc) can be customized, with any logic, different for different OS, build systems, etc.

```python
global import ConanFile

class Pkg(ConanFile):
    settings = "os", "compiler", "arch", "build_type"
    def package_info(self):
        if not self.in_local_cache:
            if self.settings.compiler == "Visual Studio":
                # NOTE: Use the real layout used in your VS projects, this is just an example
                self.cpp_info.libdirs = ["%s_%s" % (self.settings.build_type, self.settings.arch)]
```

That will define the libraries directories to `path/to/local/dev/libcool/Release_x86_64`, for example. That is only an example, the real layout used by VS would be different.

Layout files

Instead of changing the recipe file to match the local layout, it’s possible to define the layout in a separate file. This is especially useful if you have a large number of libraries with the same structure so you can write it once and use it for several packages.

Layout files are ini files, but before parsing them Conan uses the Jinja2 template engine passing the `settings`, `options` and current `reference` objects, so you can add logic to the files:

```ini
[includedirs]
src/core/include
src/cmp_a/include

[libdirs]
build/{{settings.build_type}}/{{settings.arch}}

[bindirs]
{% if options.shared %}
build/{{settings.build_type}}/shared
{% else %}
build/{{settings.build_type}}/static
{% endif %}
```

(continues on next page)
You can have a look at the Jinja2 documentation to know more about its powerful syntax.

This file can use the package reference to customize logic for a specific package:

```
[includedirs]
src/include

[cool/version@user/dev:includedirs]
src/core/include
```

This layout will define the `src/core/include` include directory for the `cool` package, and `src/include` for other packages in editable mode.

In every case the directories that will be affected by the editable mode will be `includedirs`, `libdirs`, `bindirs`, `resdirs`, `srcdirs` and `builddirs`, all of them declared in the `cpp_info` dictionary; the rest of values in that dictionary won’t be modified. So `cflags`, `defines`, library names in `libs` defined in `package_info()` will still be used.

By default all folders paths are relative to the directory where the `conanfile.py` of the editable package is (which is the path used to create the link), though they also allow absolute paths.

### Specifying layout files

Layout files are specified in the `conan editable add` command, as an extra argument:

```
$ conan editable add . cool/version@user/dev --layout=win_layout
```

That `win_layout` file will be first looked for relative to the current directory (the path can be absolute too). If it is found, that will be used. It is possible to add those layouts in the source repositories, so they are always easy to find after a clone.

If the specified layout is not found relative to the current directory, it will be looked for in the conan cache, in the `.conan/layouts` folder. This is very convenient to have a single definition of layouts that can be shared with the team and installed with `conan config install`.

If no argument is specified, the `conan editable add` command will try to use a `.conan/layouts/default` layout from the local cache.

You can switch layout files by passing a different argument to new calls to `conan editable add`.

### Evaluation order and priority

It is important to understand the evaluation order and priorities regarding the definitions of layouts:

- The first thing that will always execute is the recipe `package_info()`. That will define the flags, definitions, as well as some values for the layout folders: `includedirs`, `libdirs`, etc.
- If a layout file is defined, either explicitly or using the implicit `.conan/layouts/default`, conan will look for matches, based on its package reference.
• If a match is found, either because of global definitions like [includedirs] or because a match like [pkg/version@user/channel:includedirs], then the layout folders (includedirs, libdirs, resdirs, builddirs, bindirs), will be invalidated and replaced by the ones defined in the file.

• If a specific match like [pkg/version@user/channel:includedirs] is found, it is expected to have defined also its specific [pkg/version@user/channel:libdirs], etc. The global layout folders specified without package reference won’t be applied once a match is found.

• It no match is found, the original values for the layout folders defined in package_info() will be respected.

• The layout file to be used is defined at conan editable add time. If a .conan/layouts/default file is added after the conan editable add, it will not be used at all.

7.2.3 Using a package in editable mode

Once a reference is in editable mode it is used system wide (for every set of settings and options) by Conan (by every Conan client that uses the same cache), no changes are required in the consumers. Every conan install command that requires our editable cool/version@user/dev package will use the paths to the local directory and the changes made to this project will be taken into account by the packages using its headers or linking against it.

To summarize, consumption of packages in editable mode is transparent to their consumers. To try that it is working, the following flow should work:

• Get sources of cool/version@user/dev: git/svn clone... && cd folder

• Put package in editable mode: conan editable add . cool/version@user/dev --layout=mylayout

• Work with it and build using any tool. Check that your local layout is reflected in the layout file mylayout specified in the previous step.

• Go to the consumer project: CoolApp

• Build it using any local flow: conan install and build

• Go back to cool/version@user/dev source folder, do some changes, and just build. No Conan commands necessary

• Go to the consumer project: CoolApp and rebuild. It should get the changes from the cool library.

In that way, it is possible to be developing both the cool library and the CoolApp application, at the same time, without any Conan command.

Note: When a package is in editable mode, most of the commands will not work. It is not possible to conan upload, conan export or conan create when a package is in editable mode.

7.2.4 Revert the editable mode

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

```bash
$ conan editable remove cool/version@user/dev
```

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 consuming an editable package in its graph upstreams can generate binaries and packages 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.

7.3 Workspaces

Warning: This is an experimental feature. This is actually a preview of the feature, with the main goal of receiving feedbacks and improving it. Consider the file formats, commands and flows to be unstable and subject to changes in the next releases.

Sometimes, it is necessary to work simultaneously on more than one package. In theory, each package should be a “work unit”, and developers should be able to work on them in isolation. But sometimes, some changes require modifications in more than one package at the same time. The local development flow can help, but it still requires using `export-pkg` to put the artifacts in the local cache, where other packages under development will consume them.

The Conan workspaces allow to have more than one package in user folders, and have them directly use other packages from user folders without needing to put them in the local cache. Furthermore, they enable incremental builds on large projects containing multiple packages.

Let’s introduce them with a practical example; the code can be found in the conan examples repository:

```
$ git clone https://github.com/conan-io/examples.git
$ cd features/workspace/cmake
```

Note that this folder contains two files `conanws_gcc.yml` and `conan_vs.yml`, for gcc (Makefiles, single-configuration build environments) and for Visual Studio (MSBuild, multi-configuration build environment), respectively.

7.3.1 Conan workspace definition

Workspaces are defined in a yaml file, with any user defined name. Its structure is:

```
edittables:
    say/0.1@user/testing:
      path: say
    hello/0.1@user/testing:
      path: hello
    chat/0.1@user/testing:
      path: chat
layout: layout_gcc
workspace_generator: cmake
root: chat/0.1@user/testing
```

The first section `editables` defines the mapping between package references and relative paths. Each one is equivalent to a `conan editable add` command (Do NOT do this – it is not necessary. It will be automatically done later. Just to understand the behavior):

```
$ conan editable add say say/0.1@user/testing --layout=layout_gcc
$ conan editable add hello hello/0.1@user/testing --layout=layout_gcc
$ conan editable add chat chat/0.1@user/testing --layout=layout_gcc
```
The main difference is that this Editable state is only temporary for this workspace. It doesn’t affect other projects or packages, which can still consume these say, hello, chat packages from the local cache.

Note that the layout: layout_gcc declaration in the workspace affects all the packages. It is also possible to define a different layout per package, as:

```yaml
editables:
  say/0.1@user/testing:
    path: say
    layout: custom_say_layout
```

Layout files are explained in Editable layout files and in the Packages in editable mode sections.

The workspace_generator defines the file that will be generated for the top project. The only supported value so far is cmake and it will generate a `conanworkspace.cmake` file that looks like:

```cmake
set(PACKAGE_say_SRC "<path>/examples/workspace/cmake/say/src")
set(PACKAGE_say_BUILD "<path>/examples/workspace/cmake/say/build/Debug")
set(PACKAGE_hello_SRC "<path>/examples/workspace/cmake/hello/src")
set(PACKAGE_hello_BUILD "<path>/examples/workspace/cmake/hello/build/Debug")
set(PACKAGE_chat_SRC "<path>/examples/workspace/cmake/chat/src")
set(PACKAGE_chat_BUILD "<path>/examples/workspace/cmake/chat/build/Debug")

macro(conan_workspace_subdirectories)
  add_subdirectory(${PACKAGE_say_SRC} ${PACKAGE_say_BUILD})
  add_subdirectory(${PACKAGE_hello_SRC} ${PACKAGE_hello_BUILD})
  add_subdirectory(${PACKAGE_chat_SRC} ${PACKAGE_chat_BUILD})
endmacro()
```

This file can be included in your user-defined `CMakeLists.txt` (this file is not generated). Here you can see the `CMakeLists.txt` used in this project:

```cmake
cmake_minimum_required(VERSION 3.0)
project(WorkspaceProject)
include(${CMAKE_BINARY_DIR}/conanworkspace.cmake)
canon_workspace_subdirectories()
```

The root: chat/0.1@user/testing defines which is the consumer node of the graph, typically some kind of executable. You can provide a comma separated list of references, as a string, or a yaml list (abbreviated or full as yaml items). All the root nodes will be in the same dependency graph, leading to conflicts if they depend on different versions of the same library, as in any other Conan command.

```yaml
editables:
  say/0.1@user/testing: 
    path: say
  hello/0.1@user/testing: 
    path: hello
  chat/0.1@user/testing: 
    path: chat

root: chat/0.1@user/testing, say/0.1@user/testing # or
root: ["HelloA/0.1@lasote/stable", "HelloB/0.1@lasote/stable"] # or
root:
```

(continues on next page)
7.3.2 Single configuration build environments

There are some build systems, like Make, that require the developer to manage different configurations in different build folders, and switch between folders to change configuration. The file described above is `conan_gcc.yml` file, which defines a Conan workspace that works for a CMake based project for MinGW/Unix Makefiles gcc environments (working for apple-clang or clang would be very similar, if not identical).

Let's use it to install this workspace:

```
$ mkdir build_release && cd build_release
$ conan workspace install ../conanws_gcc.yml --profile=my_profile
```

Here we assume that you have a `my_profile` profile defined which would use a single-configuration build system (like Makefiles). The example is tested with gcc in Linux, but working with apple-clang with Makefiles would be the same). You should see something like:

```
Configuration:
[settings]
... build_type=Release
compiler=gcc
compiler.libcxx=libstdc++
compiler.version=4.9
...

Requirements
 chat/0.1@user/testing from user folder - Editable
 hello/0.1@user/testing from user folder - Editable
 say/0.1@user/testing from user folder - Editable

Packages
 chat/0.1@user/testing:df2c4f472519597d4834538680abd57ff9 - Editable
 hello/0.1@user/testing:b0e473ad8697d6069797b921517628bba8b5901 - Editable
 say/0.1@user/testing:80faec7955dcbba29246085ff8d64a765db3b414f - Editable

say/0.1@user/testing: Generator cmake created conanbuildinfo.cmake
...
hello/0.1@user/testing: Generator cmake created conanbuildinfo.cmake
...
chat/0.1@user/testing: Generator cmake created conanbuildinfo.cmake
...
```

These `conanbuildinfo.cmake` files have been created in each package build/Release folder, as defined by the `layout_gcc` file:

```
# This helps to define the location of CMakeLists.txt within package
[source_folder]
src

# This defines where the conanbuildinfo.cmake will be written to
[build_folder]
build/{{settings.build_type}}
```
Now we can configure and build our project as usual:

$ cmake .. -G "Unix Makefiles" -DCMAKE_BUILD_TYPE=Release
$ conan --build . # or just $ make
$ ../chat/build/Release/app
Release: Hello World!
Release: Hello World!
Release: Hello World!

Now, go do a change in some of the packages, for example the “say” one, and rebuild. See how it does an incremental build (fast).

Note that nothing will really be installed in the local cache, all the dependencies are resolved locally:

$ conan search say
There are no packages matching the 'say' pattern

Note: The package conanfile.py recipes do not contain anything special, they are standard recipes. But the packages CMakeLists.txt have defined the following:

conan_basic_setup(NO_OUTPUT_DIRS)

This is because the default conan_basic_setup() does define output directories for artifacts such as bin, lib, etc, which is not what the local project layout expects. You need to check and make sure that your build scripts and recipe matches both the expected local layout (as defined in layout files), and the recipe package() method logic.

Building for debug mode is done in its own folder:

$ cd .. $ $ mkdir build_debug $ $ cd build_debug
$ conan workspace install ../conanws_gcc.yml --profile=my_gcc_profile -s build_→_type=Debug
$ conan --build . # or just $ make
$ ../chat/build/Debug/app
Debug: Bye World!
Debug: Bye World!
Debug: Bye World!

7.3.3 Multi configuration build environments

Some build systems, like Visual Studio (MSBuild), use “multi-configuration” environments. That is, even if the project is configured just once you can switch between different configurations (like Debug/Release) directly in the IDE and build there.

The above example uses the Conan cmake generator, that creates a single conanbuildinfo.cmake file. This is not a problem if we have our configurations built in different folders. Each one will contain its own conanbuildinfo.cmake. For Visual Studio that means that if we wanted to switch from Debug<->Release, we should issue a new conan workspace install command with the right -s build_type and do a clean build, in order to get the right dependencies.

Conan has the cmake_multi generator, that allows this direct switch of Debug/Release configuration in the IDE. The conanfile.py recipes they have defined the cmake generator, so the first step is to override that in our conanws_vs.yml file:
Note the generators: `cmake_multi` line, that will define the generators to be used by our workspace packages. Also, our `CMakeLists.txt` should take into account that now we won’t have a `conanbuildinfo.cmake` file, but a `conanbuildinfo_multi.cmake` file. See for example the `hello/src/CMakeLists.txt` file:

```cmake
project(Hello)
if(EXISTS ${CMAKE_CURRENT_BINARY_DIR}/conanbuildinfo_multi.cmake)
  include(${CMAKE_CURRENT_BINARY_DIR}/conanbuildinfo_multi.cmake)
else()
  include(${CMAKE_CURRENT_BINARY_DIR}/conanbuildinfo.cmake)
endif()
conan_basic_setup(NO_OUTPUT_DIRS)
add_library(hello hello.cpp)
conan_target_link_libraries(hello)
```

The last `conan_target_link_libraries(hello)` is a helper that does the right linking with Debug/Release libraries (also works when using cmake targets).

Make sure to install both Debug and Release configurations straight ahead, if we want to later switch between them in the IDE:

```
$ mkdir build && cd build
$ conan workspace install ../conanws_vs.yml
$ conan workspace install ../conanws_vs.yml -s build_type=Debug
$ cmake .. -G "Visual Studio 15 Win64"
```

With those commands you will get a Visual Studio solution, that you can open, select the `app` executable as StartUp project, and start building, executing, debugging, switching from Debug and Release configurations freely from the IDE, without needing to issue further Conan commands.

You can check in the project folders, how the following files have been generated:

```
hello
|-- build
    |-- conanbuildinfo_multi.cmake
    |-- conanbuildinfo_release.cmake
    |-- conanbuildinfo_debug.cmake
```

Note that they are not located in `build/Release` and `build/Debug` subfolders; that is because of the multi-config environment. To account for that the `layout_vs` define the `[build_folder]` not as `build/{settings.build_type}` but just as:

```
[build_folder]
build
```
7.3.4 Out of source builds

The above examples are using a build folder in-source of the packages in editable mode. It is possible to define out-of-source builds layouts, using relative paths and the `reference` argument. The following layout definition could be used to locate the build artifacts of an editable package in a sibling `build/<package-name>` folder:

```
[build_folder]
  ../build/{{reference.name}}/{{settings.build_type}}

[includedirs]
  src

[libdirs]
  ../build/{{reference.name}}/{{settings.build_type}}/lib
```

7.3.5 Notes

Note that this way of developing packages shouldn’t be used to create the final packages (you could try to use `conan export-pkg`), but instead, a full package creation with `conan create` (best in CI) is recommended.

So far, only the CMake super-project generator is implemented. A Visual Studio one is being considered, and seems feasible, but not yet available.

**Important:** We really want your feedback. Please submit any issues to https://github.com/conan-io/conan/issues with any suggestion, problem, idea, and using [workspaces] prefix in the issue title.
With conan it is possible to package and deploy applications. It is also possible that these applications are also dev-tools, like compilers (e.g. MinGW), or build systems (e.g. CMake).

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

### 8.1 Running and deploying packages

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

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

There are different approaches:

#### 8.1.1 Using virtual environments

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

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

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

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

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

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

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

The `virtualrunenv` generator generates files that add the package’s default binary locations to the necessary paths:

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

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

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

This will generate a few files that can be called to activate and deactivate the required environment variables

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

### 8.1.2 Imports

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

### 8.1.3 Deployable packages

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

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

And run `conan create`

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

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

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

The deploy will create a `deploy_manifest.txt` file with the files that have been deployed.
Sometimes it is useful to adjust the package ID of the deployable package in order to deploy it regardless of the compiler it was compiled with:

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

See also:
Read more about the `deploy()` method.

### 8.1.4 Running from packages

If a dependency has an executable that we want to run in the conanfile, it can be done directly in code using the `run_environment=True` argument. It internally uses a `RunEnvironment` helper. For example, if we want to execute the `greet` app while building the `Consumer` package:

```python
from conans import ConanFile, tools, RunEnvironment
class ConsumerConan(ConanFile):
    name = "Consumer"
    version = "0.1"
    settings = "os", "compiler", "build_type", "arch"
    requires = "Hello/0.1@user/testing"
    def build(self):
        self.run("greet", run_environment=True)
```

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

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

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

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

Note that this might not be enough if shared libraries exist. Using the `run_environment=True` helper above is a more complete solution.

Finally, there is another approach: the package containing the executable can add its `bin` folder directly to the `PATH`. In this case the `Hello` package conanfile would contain:

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

We may also define `DYLD_LIBRARY_PATH` and `LD_LIBRARY_PATH` if they are required for the executable. The consumer package is simple, as the `PATH` environment variable contains the `greet` executable:

```python
def build(self):
    self.run("greet")
```
8.1.5 Runtime packages and re-packaging

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

```python
from conans import ConanFile

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

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

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

This recipe has the following characteristics:

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

To create and upload this package to a remote:

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

Installing and running this package can be done using any of the methods presented above. For example:

```
$ conan install HelloRun/0.1@user/testing -g virtualrunenv
# You can specify the remote with -r=my-remote
# It will not install Hello/0.10...
$ activate_run.sh # $ source activate_run.sh in Unix/Linux
$ greet
> Hello World!
$ deactivate_run.sh # $ source deactivate_run.sh in Unix/Linux
```

8.2 Creating conan packages to install dev tools

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

These settings are different from os and arch. These mean where the built software by the Conan recipe will run. When we are packaging a tool, it usually makes no sense, because we are not building any software, but it makes sense if you are cross building software.
We recommend the use of os_build and arch_build settings instead of os and arch if you are packaging a tool involved in the building process, like a compiler, a build system etc. If you are building a package to be run on the host system you can use os and arch.

A Conan package for a tool follows always a similar structure. This is a recipe for packaging the nasm tool for building assembler:

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

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

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

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

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

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

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

There are some remarkable things in the recipe:

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

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

• `source()` method is missing. That’s because when you compile a library, the source code is always the same for all the generated packages. In this case we are downloading the binaries, so we do it in the build method to download the appropriate zip file according to each combination of settings/options. Instead of actually building the tools, we just download them. Of course, if you want to build it from source, you can do it too by creating your own package recipe.

• The `package_info()` method uses the new `self.env_info` object. With `self.env_info` the package can declare environment variables that will be set automatically before `build()`, `package()`, `source()` and `imports()` methods of a package requiring this build tool. This is a convenient method to use these tools without having to manipulate the system path.

### 8.2.1 Using the tool packages in other recipes

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

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

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

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

        files = {
            ...
        }

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

        ...
```

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

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

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

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

    def build(self):
```

(continues on next page)
cmake = CMake(self)
...

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

Note package_id() deletes unneeded information for the computation of the package ID and includes the build settings os_build and arch_build that are excluded by default. Read more about self.info.include_build_settings() in the reference section.

8.2.2 Using the tool packages in your system

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

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

   $ mkdir my_cpp_environ
   $ cd my_cpp_environ

2. Create a conanfile.txt file:

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

   [generators]
  virtualenv

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

3. Install them:

   $ conan install .

4. Activate the virtual environment in your shell:

   $ activate
   (my_cpp_environ)$

5. Check that the tools are in the path:

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

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

   (my_cpp_environ)$ cmake --version
6. You can deactivate the virtual environment with the deactivate.bat script

```
(my_cpp.environ)$ deactivate
```

### 8.3 Build requirements

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

In short:

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

To address these needs Conan implements **build_requires**.

#### 8.3.1 Declaring build requirements

Build requirements can be declared in profiles, like:

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

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

If a pattern like MyPkg* is specified, the declared build requirements will only be applied to packages matching that pattern. Tool5 will not be applied to Zlib for example, but it will be applied to MyPkgZlib.
The special case of a **consumer** conanfile (without name or version) it is impossible to match with a pattern, so it is handled with the special character `&`:

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

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

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

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

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

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

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

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

### 8.3.2 Properties of build requirements

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

- They will only be retrieved and installed if some package that has to be built from sources and matches the declared pattern. Otherwise, they will not even be checked for existence.
- Options and environment variables declared in the profile as well as in the command line will affect the build requirements for packages. In that way, you can define, for example, for the `cmake_installer/0.1` package which CMake version will be installed.
- Build requirements will be activated for matching packages via the `deps_cpp_info` and `deps_env_info` members. So, include directories, library names, compile flags (CFLAGS, CXXFLAGS, LINKFLAGS), sysroot, etc. will be applied from the build requirement’s package `self.cpp_info` values. The same for `self.env_info`: variables such as PATH, PYTHONPATH, and any other environment variables will be applied to the matching patterns and activated as environment variables.
- Build requirements can also be transitive. They can declare their own requirements, both normal requirements and their own build requirements. Normal logic for dependency graph resolution applies, such as conflict resolution and dependency overriding.
- Each matching pattern will produce a different dependency graph of build requirements. These graphs are cached so that they are only computed once. If a build requirement applies to different packages with the same configuration it will only be installed once (same behavior as normal dependencies - once they are cached locally, there is no need to retrieve or build them again).
Build requirements do not affect the binary package ID. If using a different build requirement produces a different binary, you should consider adding an option or a setting to model that (if not already modeled).

Can also use version-ranges, like `Tool/[>0.3]@user/channel`.

Build requirements are not listed in `conan info` nor are represented in the graph (with `conan info --graph`).

### 8.3.3 Testing libraries

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

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

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

And the package `CMakeLists.txt`:

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

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

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

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

```
$ conan install .
```

But if the following profile is defined:

```
Listing 2: mytest_profile

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

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

```
$ conan install . --profile=mytest_profile
```
8.3.4 Common python code

**Warning:** This way of reusing python code has been superseded by `python_requires`. Please check [Python requires: reusing code](https://conan.io/). The same technique can even be used to inject and reuse python code in the package recipes, without having to declare dependencies to such python packages.

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

```python
import os
from conans import ConanFile

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

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

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

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

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

such package can be reused in other recipes like this:

```python
def build(self):
    self.run("mytool")
    import mypythontool
    self.output.info(mypythontool.hello_world())
```
This section provides an introduction to important productivity and useful features of Conan:

### 9.1 Use conanfile.py for consumers

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

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

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

[generators]
gcc
cmake
txt

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

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

The equivalent conanfile.py file is:

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

def imports(self):
    self.copy("*.dll", dst="bin", src="bin") # From bin to bin
    self.copy("*.dylib*", dst="bin", src="lib") # From lib to bin
```
Note that this `conanfile.py` doesn’t have a name, version, or `build()` or `package()` method, as it is not creating a package. They are not required.

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

```
$ conan install ..
```

### 9.1.1 conan build

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

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

```python
from conans import ConanFile, CMake

class PocoTimerConan(ConanFile):
    settings = "os", "compiler", "build_type", "arch"
    requires = "Poco/1.7.8p3@pocoproject/stable"
    generators = "cmake", "gcc", "txt"
    default_options = {"Poco:shared": True, "OpenSSL:shared": True}
    
    def imports(self):
        self.copy("*.dll", dst="bin", src="bin") # From bin to bin
        self.copy("*.dylib*", dst="bin", src="lib") # From lib to bin
    
    def build(self):
        cmake = CMake(self)
        cmake.configure()
        cmake.build()
```

Then execute, from your project root:

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

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

This `conan build` will use the settings used in the `conan install` which have been cached in the local `conaninfo.txt` and file in your build folder. This simplifies the process and reduces the errors of mismatches between the installed packages and the current project configuration. Also, the `conanbuildinfo.txt` file contains all the needed information obtained from the requirements: `deps_cpp_info`, `deps_env_info`, `deps_user_info objects`.

If you want to build your project for `x86` or another setting just change the parameters passed to `conan install`:

```
$ conan install . --install-folder build_x86 -s arch=x86
$ conan build . --build-folder build_x86
```

Implementing and using the `conanfile.py` `build()` method ensures that we always use the same settings both in the installation of requirements and the build of the project, and simplifies calling the build system.
9.1.2 Other local commands

Conan implements other commands that can be executed locally over a consumer `conanfile.py` which is in user space, not in the local cache:

- `conan source <path>`: Execute locally the `conanfile.py source()` method.
- `conan package <path>`: Execute locally the `conanfile.py package()` method.

These commands are mostly used for testing and debugging while developing a new package, before exporting such package recipe into the local cache.

See also:

Check the section *Reference/Commands* to find out more.

9.2 Conditional settings, options and requirements

Remember, in your `conanfile.py` you also have access to the options of your dependencies, and you can use them to:

- Add requirements dynamically
- Change values of options

The `configure` method might be used to hardcode dependencies options values. It is strongly discouraged to use it to change the settings values. Please remember that `settings` are a configuration *input*, so it doesn’t make sense to modify it in the recipes.

Also, for options, a more flexible solution is to define dependencies options values in the `default_options`, not in the `configure()` method, as this would allow to override them. Hardcoding them in the `configure()` method won’t allow that and thus won’t easily allow conflict resolution. Use it only when it is absolutely necessary that the package dependencies use those options.

Here is an example of what we could do in our `configure` method:

```
...  
requires = "Poco/1.9.0@pocoproject/stable"  # We will add OpenSSL dynamically "OpenSSL/
˓→1.0.2d@lasote/stable"
...

def configure(self):
    # We can control the options of our dependencies based on current options
    self.options["OpenSSL"].shared = self.options.shared

    # Maybe in windows we know that OpenSSL works better as shared (false)
    if self.settings.os == "Windows":
        self.options["OpenSSL"].shared = True

    # Or adjust any other available option
    self.options["Poco"].other_option = "foo"

    # We could check the presence of an option
    if "shared" in self.options:
        pass

def requirements(self):
    # Or add a new requirement!
```

(continues on next page)
9.2.1 Constrain settings and options

Sometimes there are libraries that are not compatible with specific settings like libraries that are not compatible with an architecture, or options that only make sense for an operating system. It can also be useful when there are settings under development.

There are two approaches for this situation:

- **Use configure() to raise an error for non-supported configurations:**
  
  This approach is the first one evaluated when Conan loads the recipe so it is quite handy to perform checks of the input settings. It relies on the set of possible settings inside your settings.yml file, so it can be used to constrain any recipe.

  ```python
  from conans.errors import ConanInvalidConfiguration
  ...
  def configure(self):
    if self.settings.os == "Windows":
      raise ConanInvalidConfiguration("This library is not compatible with Windows")
  
  Tip: Use the Invalid configuration exception to make Conan return with a special error code. This will indicate that the configuration used for settings or options is not supported.

  This same method is also valid for options and config_options() method and it is commonly used to remove options for one setting:

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

- **Constrain settings inside a recipe:**

  This approach constrains the settings inside a recipe to a subset of them, and it is normally used in recipes that are never supposed to work out of the restricted settings.

  ```python
  from conans import ConanFile

  class MyConan(ConanFile):
    name = "myconanlibrary"
    version = "1.0.0"
    settings = {"os": None, "build_type": None, "compiler": None, "arch": ["x86_64"]}
  ```

  The disadvantage of this is that possible settings are hardcoded in the recipe, and in case new values are used in the future, it will require the recipe to be modified explicitly.

  **Important:** Note: the use of the None value in the os, compiler and build_type settings described
above will allow them to take the values from settings.yml file

We strongly recommend the use of the first approach whenever it is possible, and use the second one only for those cases where a stronger constrain is needed for a particular recipe.

See also:
Check the reference section configure(), config_options() to find out more.

### 9.3 Version ranges

Version range expressions are supported, both in `conanfile.txt` and in `conanfile.py` requirements.

The syntax uses brackets. The square brackets are the way to inform Conan that is a version range. Otherwise, versions are plain strings. They can be whatever you want them to be (up to limitations of length and allowed characters).

```python
class HelloConan(ConanFile):
  requires = "Pkg/]>1.0 <1.8@user/stable"
```

So when specifying `Pkg/[expression]@user/stable`, it means that `expression` will be evaluated as a version range. Otherwise, it will be understood as plain text, so `requires = "Pkg/version@user/stable"` always means to use the version version literally.

There are some packages that do not follow semver. A popular one would be the OpenSSL package with versions as 1.0.2n. They cannot be used with version-ranges. To require such packages you always have to use explicit versions (without brackets).

The process to manage plain versions vs version-ranges is also different. The second one requires a “search” in the remote, which is orders of magnitude slower than direct retrieval of the reference (plain versions). Take it into account if you plan to use it for very large projects.

Expressions are those defined and implemented by [https://pypi.org/project/node-semver/](https://pypi.org/project/node-semver/). Accepted expressions would be:

```
[>1.1 <2.1]    # In such range
[2.8]          # equivalent to =2.8
[~3.0]         # compatible, according to semver
[>1.1 || 0.8]  # conditions can be OR'ed
[1.2.7 || >=1.2.9 <2.0.0]  # This range would match the versions 1.2.7, 1.2.9, and 1.4.6, but not the versions 1.2.8 or 2.0.0.
```

There are two options for the version range:

- **loose=True|False**: As default Conan uses the `loose=True` mode. When using `loose=False` the loose mode is deactivated, so that only valid Semantic Versioning strings are accepted.

- **include_prerelease=True|False**: Default is `include_prerelease=False`. When set to `include_prerelease=True`, it will include prerelease versions in the search range.

```
[>1.1 <2.1, include_prerelease=True] # Would e.g. accept "2.0.0-pre.1" as match
[~1.2.3, loose=False]               # Would only accept correct Semantic Versioning strings.
  # E.g. version "1.2.3.4" would not be accepted.
[~1.2.3, loose=False, include_prerelease=True] # Both options can be used for the same version range.
```
Version range expressions are evaluated at the time of building the dependency graph, from downstream to upstream dependencies. No joint-compatibility of the full graph is computed. Instead, version ranges are evaluated when dependencies are first retrieved.

This means, that if a package A depends on another package B (A->B), and A has a requirement for C/\[>1.2 <1.8\], this requirement is evaluated first and it can lead to get the version C/1.7. If package B has the requirement to C/\[>1.3 <1.6\], this one will be overwritten by the downstream one, it will output a version incompatibility error. But the “joint” compatibility of the graph will not be obtained. Downstream packages or consumer projects can impose their own requirements to comply with upstream constraints. In this case a override dependency to C/\[>1.3 <1.6\] can be easily defined in the downstream package or project.

The order of search for matching versions is as follows:

• First, the local conan storage is searched for matching versions, unless the --update flag is provided to conan install.
• If a matching version is found, it is used in the dependency graph as a solution.
• If no matching version is locally found, it starts to search in the remotes, in order. If some remote is specified with -r=remote, then only that remote will be used.
• If the --update parameter is used, then the existing packages in the local conan cache will not be used, and the same search of the previous steps is carried out in the remotes. If new matching versions are found, they will be retrieved, so subsequent calls to install will find them locally and use them.

9.4 Build policies

By default, conan install command will search for a binary package (corresponding to our settings and defined options) in a remote. If it’s not present the install command will fail.

As previously demonstrated, we can use the --build option to change the default conan install behavior:

• --build some_package will build only “some_package”.
• --build missing will build only the missing requires.
• --build will build all requirements from sources.
• --build outdated will try to build from code if the binary is not built with the current recipe or when missing binary package.
• --build cascade will build from code all the nodes with some dependency being built (for any reason). Can be used together with any other build policy. Useful to make sure that any new change introduced in a dependency is incorporated by building again the package.
• --build pattern* will build only the packages with the reference starting with “pattern”.

With the build_policy attribute in the conanfile.py the package creator can change the default Conan’s build behavior. The allowed build_policy values are:

• missing: If no binary package is found, Conan will build it without the need to invoke Conan install with --build missing option.
• always: The package will be built always, retrieving each time the source code executing the “source” method.

```python
class PocoTimerConan(ConanFile):
    settings = "os", "compiler", "build_type", "arch"
    requires = "Poco/1.7.8p3@pocoproject/stable" # comma-separated list of
    # requirements
```

(continues on next page)
generators = "cmake", "gcc", "txt"
default_options = {"Poco:shared": True, "OpenSSL:shared": True}
build_policy = "always" # "missing"

These build policies are especially useful if the package creator doesn’t want to provide binary package; for example, with header only libraries.

The always policy will retrieve the sources each time the package is installed, so it can be useful for providing a “latest” mechanism or ignoring the uploaded binary packages.

The package pattern can be referred as a case-sensitive fnmatch pattern of the package name or the full package reference. e.g: --build Poco, --build Poc*, --build zlib/*@conan/*, --build *@conan/stable or --build zlib/1.2.11@conan/stable.

9.5 Environment variables

There are several use cases for environment variables:

- Conan global configuration environment variables (e.g. CONAN_COMPRESSION_LEVEL). They can be configured in conan.conf or as system environment variables, and control Conan behavior.
- Package recipes can access environment variables to determine their behavior. A typical example would be when launching CMake. It will check for CC and CXX environment variables to define the compiler to use. These variables are mostly transparent for Conan, and just used by the package recipes.
- Environment variables can be set in different ways:
  - global, at the OS level, with `export VAR=Value` or in Windows `SET VAR=Value`.
  - In the Conan command line: `conan install -e VAR=Value`.
  - In profile files.
  - In package recipes in the `self.env_info` field, so they are activated for dependent recipes.

9.5.1 Defining environment variables

You can use profiles to define environment variables that will apply to your recipes. You can also use `-e` parameter in `conan install`, `conan info` and `conan create` commands.

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

If you want to override an environment variable that a package has inherited from its requirements, you can use either profiles or `-e` to do it:

```
conan install . -e MyPackage:PATH=/other/path
```

If you want to define an environment variable, but you want to append the variables declared in your requirements, you can use the `[]` syntax:

```
$ conan install . -e PYTHONPATH=[/other/path]
```

This way the first entry in the PYTHONPATH variable will be `/other/path`, but the PYTHONPATH values declared in the requirements of the project will be appended at the end using the system path separator.
9.5.2 Automatic environment variables inheritance

If your dependencies define some `env_info` variables in the `package_info()` method, they will be automatically applied before calling the consumer `conanfile.py` methods `source()`, `build()`, `package()`, and `imports()`. You can read more about `env_info` object [here](#).

For example, if you are creating a package for a tool, you can define the variable `PATH`:

```python
class ToolExampleConan(ConanFile):
    name = "my_tool_installer"
    ...

def package_info(self):
    self.env_info.path.append(os.path.join(self.package_folder, "bin"))
```

If another Conan recipe requires the `my_tool_installer` in the `source()`, `build()`, `package()`, and `imports()`, the bin folder of the `my_tool_installer` package will be automatically appended to the system PATH. If `my_tool_installer` packages an executable called `my_tool_executable` in the `bin` of the package folder, we can directly call the tool because it will be available in the path:

```python
class MyLibExample(ConanFile):
    name = "my_lib_example"
    ...

def build(self):
    self.run("my_tool_executable some_arguments")
```

You could also set `CC`, `CXX` variables if we are packing a compiler to define what compiler to use or any other environment variable. Read more about tool packages [here](#).

9.6 Virtual Environments

Conan offers three special Conan generators to create virtual environments:

- `virtualenv`: Declares the `self.env_info` variables of the requirements.
- `virtualbuildenv`: Special build environment variables for autotools/visual studio.
- `virtualrunenv`: Special environment variables to locate executables and shared libraries in the requirements.

These virtual environment generators create two executable script files (.sh or .bat depending on the current operating system), one to activate the virtual environment (set the environment variables) and one to deactivate it.

You can aggregate two or more virtual environments, that means that you can activate a `virtualenv` and then activate a `virtualrunenv` so you will have available the environment variables declared in the `env_info` object of the requirements plus the special environment variables to locate executables and shared libraries.

9.6.1 Virtualenv generator

Conan provides a `virtualenv` generator, able to read from each dependency the `self.env_info` variables declared in the `package_info()` method and generate two scripts “activate” and “deactivate”. These scripts set/unset all env variables in the current shell.

**Example:**

The recipe of `cmakeInstaller/3.9.0@conan/stable` appends to the PATH variable the package folder/bin.
You can check existing CMake conan package versions in conan-center with:

```bash
$ conan search cmake* -r=conan-center
```

In the `bin` folder there is a `cmake` executable:

```python
def package_info(self):
    self.env_info.path.append(os.path.join(self.package_folder, "bin"))
```

Let’s prepare a virtual environment to have cmake available in the path. Open `conanfile.txt` and change (or add) `virtualenv` generator:

```ini
[requires]
cmake_installer/3.9.0@conan/stable

[generators]
virtualenv
```

Run `conan install`:

```bash
$ conan install .
```

You can also avoid the creation of the `conanfile.txt` completely and directly do:

```bash
$ conan install cmake_installer/3.9.0@conan/stable -g=virtualenv
```

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

```bash
$ source activate.sh  # Windows: activate.bat without the source
$ cmake --version
```

Two sets of scripts are available for Windows - `activate.bat/deactivate.bat` and `activate.ps1/deactivate.ps1` if you are using powershell. Deactivate the virtual environment (or close the console) to restore the environment variables:

```bash
$ source deactivate.sh  # Windows: deactivate.bat without the source
```

See also:

Read the Howto `Create installer packages` to learn more about the virtual environment feature. Check the section `Reference/virtualenv` to see the generator reference.

### 9.6.2 Virtualbuildenv environment

Use the generator `virtualbuildenv` to activate an environment that will set the environment variables for Autotools and Visual Studio.

The generator will create `activate_build` and `deactivate_build` files.

See also:

Read More about the building environment variables defined in the sections `Building with autotools` and `Build with Visual Studio`.

Check the section `Reference/virtualbuildenv` to see the generator reference.
9.6.3 Virtualrunenv generator

Use the generator `virtualrunenv` to activate an environment that will:

- Append to `PATH` environment variable every `bin` folder of your requirements.
- Append to `LD_LIBRARY_PATH` and `DYLD_LIBRARY_PATH` environment variables each `lib` folder of your requirements.

The generator will create `activate_run` and `deactivate_run` files. This generator is especially useful:

- If you are requiring packages with shared libraries and you are running some executable that needs those libraries.
- If you have a requirement with some tool (executable) and you need it in the path.

In the previous example of the `cmake_installer` recipe, even if the `cmake_installer` package doesn’t declare the `self.env_info.path` variable, using the virtualrunenv generator, the `bin` folder of the package will be available in the `PATH`. So after activating the virtual environment we could just run `cmake` in order to execute the package’s `cmake`.

See also:

- Reference/Tools/environment_append

9.7 Logging

9.7.1 How to log and debug a conan execution

You can use the `CONAN_TRACE_FILE` environment variable to log and debug several Conan command execution. Set the `CONAN_TRACE_FILE` environment variable pointing to a log file.

Example:

```bash
export CONAN_TRACE_FILE=/tmp/conan_trace.log # Or SET in windows
conan install zlib/1.2.8@lasote/stable
```

The `/tmp/conan_trace.log` file:

```json
{"_action": "COMMAND", "name": "install", "parameters": {"all": false, "build": null, "env": null, "file": null, "generator": null, "manifests": null, "manifests_interactive": null, "no_imports": false, "options": null, "package": null, "profile": null, "reference": "zlib/1.2.8@lasote/stable", "remote": null, "scope": null, "settings": null, "update": false, "verify": null, "werror": false}, "time": 1485345289.250117}
{"_action": "REST_API_CALL", "duration": 1.8255090713500977, "headers": {"Authorization": "**********, "X-Client-Anonymous-Id": "**********, "X-Client-Id": "lasote2", "X-Conan-Client-Version": "0.19.0-dev"}, "method": "GET", "time": 1485345291.092218, "url": "https://server.conan.io/v1/conans/zlib/1.2.8/lasote/stable/download_urls"}
{"_action": "DOWNLOAD", "duration": 0.4136989116668701, "time": 1485345291.506399, "url": "https://conanio-production.s3.amazonaws.com/storage/zlib/1.2.8/lasote/stable/export/conanmanifest.txt"}
{"_action": "DOWNLOAD", "duration": 0.10367798805236816, "time": 1485345291.610335, "url": "https://conanio-production.s3.amazonaws.com/storage/zlib/1.2.8/lasote/stable/export/conanfile.py"}
{"_action": "DOWNLOAD", "duration": 0.059114933013916016, "time": 1485345291.669744, "url": "https://conanio-production.s3.amazonaws.com/storage/zlib/1.2.8/lasote/stable/export/conan_export.tgz"}
```
In the traces we can see:

1. A command install execution.
2. A REST API call to get some download_urls.
3. Three files downloaded (corresponding to the previously retrieved urls).
4. DOWNLOADED_RECIPE tells us that the recipe retrieving is finished. We can see that the whole retrieve process took 2.4 seconds.
5. Conan client has computed the SHA for the needed binary package and will now retrieve it. So it will request and download the package package_id file to perform some checks like outdated binaries.
6. Another REST API call to get some more download_urls, for the package files and download them.
7. Finally we get a DOWNLOADED_PACKAGE telling us that the package has been downloaded. The download took 1.3 seconds.

If we execute conan install again:
export CONAN_TRACE_FILE=/tmp/conan_trace.log  # Or SET in windows
conan install zlib/1.2.8@lasote/stable

The /tmp/conan_trace.log file only three lines will be appended:

```json
{"_action": "COMMAND", "name": "install", "parameters": {"all": false, "build": null, "env": null, "file": null, "generator": null, "manifests": null, "manifests_interactive": null, "no_imports": false, "options": null, "package": null, "profile": null, "reference": "zlib/1.2.8@lasote/stable", "remote": null, "scope": null, "settings": null, "update": false, "verify": null, "werror": false}, "time": 1485346039.817543}

{"_action": "GOT_RECIPE_FROM_LOCAL_CACHE", "_id": "zlib/1.2.8@lasote/stable", "time": 1485346039.824949}

{"_action": "GOT_PACKAGE_FROM_LOCAL_CACHE", "_id": "zlib/1.2.8@lasote/stable:c6d75a933080ca17eb7f076813e7fb21aaa740f2", "time": 1485346039.827915}
```

1. A command install execution.
2. A GOT_RECIPE_FROM_LOCAL_CACHE because it’s already stored in local cache.
3. A GOT_PACKAGE_FROM_LOCAL_CACHE because the package is cached too.

### 9.7.2 How to log the build process

You can log your command executions `self.run` in a file named `conan_run.log` using the environment variable `CONAN_LOG_RUN_TO_FILE`.

You can also use the variable `CONAN_PRINT_RUN_COMMANDS` to log extra information about the commands being executed.

**Package the log files**

The `conan_run.log` file will be created in your `build` folder so you can package it the same way you package a library file:

```python
def package(self):
    self.copy(pattern="conan_run.log", dst="", keep_path=False)
```

### 9.8 Sharing the settings and other configuration

If you are using Conan in a company or in an organization, sometimes you need to share the `settings.yml` file, the profiles, or even the remotes or any other Conan local configuration with the team.

You can use the `conan config install`.

If you want to try this feature without affecting your current configuration, you can declare the `CONAN_USER_HOME` environment variable and point to a different directory.

Read more in the section `conan config install`.
9.9 Conan local cache: concurrency, Continuous Integration, isolation

Conan needs access to some per user configuration files, such as the `conan.conf` file that defines the basic client app configuration. By convention, this file will be located in the user home folder `~/.conan/`. This folder will also typically store the package cache in `~/.conan/data`. Even though the latter is configurable in `conan.conf`, Conan needs some place to look for this initial configuration file.

There are some scenarios in which you might want to use different initial locations for the Conan client application:

- Continuous Integration (CI) environments, in which multiple jobs can also work concurrently. Moreover, these environments would typically want to run with different user credentials, different remote configurations, etc. Note that using Continuous Integration with the same user, with isolated machine instances (virtual machines), or with sequential jobs is perfectly possible. For example, we use a lot CI cloud services of travis-ci and appveyor.
- Independent per project management and storage. If as a single developer you want to manage different projects with different user credentials and/or different remotes, you might find that having multiple independent caches makes it easier.

Using different caches is very simple. You can just define the environment variable `CONAN_USER_HOME`. By setting this variable to different paths, you have multiple conan caches, something like python “virtualenvs”. Just changing the value of `CONAN_USER_HOME`, you can switch among isolated Conan instances that will have independent package storage caches, and also different user credentials, different user default settings, and different remotes configuration.

**Note:** Use an absolute path or a path starting with `~/` (relative to user home). In Windows do not use quotes.

Windows users:

```
$ SET CONAN_USER_HOME=c:\data
$ conan install . # call conan normally, config & data will be in c:\data
```

Linux/macOS users:

```
$ export CONAN_USER_HOME=/tmp/conan
$ conan install . # call conan normally, config & data will be in /tmp/conan
```

You can now:

- Build concurrent jobs, parallel builds in Continuous Integration or locally, by just setting the variable before launching Conan commands.
- You can test locally different user credentials, default configurations, or different remotes, just by switching from one cache to another.

```
$ export CONAN_USER_HOME=/tmp/conan
$ conan search # using that /tmp/conan cache
$ conan user # using that /tmp/conan cache

$ export CONAN_USER_HOME=/tmp/conan2
$ conan search # different packages
$ conan user # can be different users

$ export CONAN_USER_HOME=/tmp/conan # just go back to use the other cache
```
9.9.1 Concurrency

Conan local cache support some degree of concurrency, allowing simultaneous creation or installation of different packages, or building different binaries for the same package. However, concurrent operations like removal of packages while creating them will fail. If you need different environments that operate totally independently, you probably want to use different Conan caches for that.

The concurrency is implemented with a Readers-Writers lock mechanism, which in turn uses fasteners library file locks to achieve multi-platform portability. As this “mutex” resource is by definition not enough to implement a Readers-Writers solution, some active-wait with time sleeps in a loop is necessary. However, this time sleeps will be rare, only sleeping when there is actually a collision and waiting on a lock.

The lock files will be stored inside each Pkg/version/user/channel folder in the local cache, in a rw file for locking the entire package, or in a set of locks (one per each different binary package, under a subfolder called locks, with each lock named with the binary ID of the package).

It is possible to disable the locking mechanism in conan.conf:

```
[general]
cache_no_locks = True
```

9.9.2 System Requirements

When system_requirements() runs, Conan creates the system_reqs folder. This folder could be created individually by package id or globally when global_system_requirements is True.

However, sometimes you want to run system_requirements() again for a specific package, so you could either remove the system_reqs.txt file for the specific package id, or you could remove system_reqs globally for the package name referred.

9.10 Package Revisions

Warning: This is an experimental feature subject to breaking changes in future releases.

The goal of the revisions feature is to achieve package immutability, the packages in a server are never overwritten.

Note: This is the first piece to achieve reproducibility: Recreate the exact dependency graph by using some mechanism like a “graph lock” file. For example, if we store a “graph lock” file for the different releases of our project, we can install the same dependencies just by using the graph lock.

IMPORTANT: The reproducibility is in the Conan roadmap and currently under development.

9.10.1 How it works

In the client

- When a recipe is exported, Conan calculates a unique ID (revision). For every change, a new recipe revision (RREV) will be calculated. By default it will use the checksum hash of the recipe manifest.

   Nevertheless, the recipe creator can explicitly declare the revision mode, it can be either scm (uses version control system or raises) or hash (use manifest hash).
When a package is created (by running `conan create` or `conan export-pkg`) a new package revision (PREV) will be calculated always using the hash of the package contents. The packages and their revisions (PREVs) belong to a concrete recipe revision (RREV). The same package ID (for example for Linux/GCC5/Debug), can have multiple revisions (PREVs) that belong to a concrete RREV.

If a client requests a reference like `lib/1.0@conan/stable`, Conan will retrieve automatically the latest revision. In the client cache there will be only one revision installed at the same time.

The revisions can be pinned when you write a reference (in the recipe requires, or in a reference in a `conan install` command…) but if you don’t specify a revision the server will retrieve the latest one.

You can specify the references in the following formats:

<table>
<thead>
<tr>
<th>Reference</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>lib/1.0@conan/stable</code></td>
<td>Latest RREV for <code>lib/1.0@conan/stable</code></td>
</tr>
<tr>
<td><code>lib/1.0@conan/stable#RREV</code></td>
<td>Specific RREV for <code>lib/1.0@conan/stable</code></td>
</tr>
<tr>
<td><code>lib/1.0@conan/stable#RREV:PACKAGE_ID</code></td>
<td>A binary package belonging to the specific RREV</td>
</tr>
<tr>
<td><code>lib/1.0@conan/stable#RREV:PACKAGE_ID#PREV</code></td>
<td>A binary package revision PREV belonging to the specific RREV</td>
</tr>
</tbody>
</table>

**In the server**

By using a new folder layout and protocol it is able to store multiple revisions, both for recipes and binary packages.

### 9.10.2 How to activate the revisions

You have to explicitly activate the feature by:

- Adding `revisions_enabled=1` in the `[general]` section of your `conan.conf` file.
- Setting the `CONAN_REVISIONS_ENABLED=1` environment variable.

Take into account that it changes the default Conan behavior. e.g:

- A client with revisions enabled will only find binary packages that belongs to the installed recipe revision. For example, If you create a recipe and run `conan create . user/channel` and then you modify the recipe and export it `conan export . user/channel`, the binary package generated in the `conan create` command doesn’t belong to the new exported recipe, so it won’t be located unless the previous recipe is recovered.
- If you generate and upload N binary packages for a recipe revision, if you upload the revision you need to generate and upload again the N binaries if you want them to be used with the new recipe.

### 9.10.3 Server support

- `conan_server` >= 1.13.
- `Artifactory` >= 6.9.
- `Bintray`.

---

**9.10. Package Revisions**
This section explains how to cross build with Conan to any platform and the Windows subsystems (Cygwin, MSYS2).

10.1 Cross-building

Cross-building is compiling a library or executable in one platform to be used in a different one.

Cross-compilation is used to build software for an alien device, such as an embedded device where you don’t have an operating system nor a compiler available. It’s also for building software for slower devices, like an Android machine, a Raspberry Pi etc.

To cross-build code you need the right toolchain.

A toolchain is basically a compiler and linker with a set of libraries matching the host platform.

10.1.1 GNU triplet convention

According to the GNU convention, there are three platforms involved in the software building:

- **Build platform**: The platform on which the compilation tools are executed
- **Host platform**: The platform on which the code will run
- **Target platform**: Only when building a compiler, this is the platform for which the compiler will generate code

When you are building code for your own machine it’s called native building, where the build and the host platforms are the same. The target platform is not defined in this situation.

When you are building code for a different platform, it’s called cross-building, where the build platform is different from the host platform. The target platform is not defined in this situation.

The use of the target platform is rarely needed. It only makes sense when you are building a compiler. For instance, when you are building on your Linux machine a GCC compiler that will run on Windows, to generate code for Android. Here, the build is your Linux computer, the host is the Windows computer and the target is Android.

10.1.2 Conan settings

From version 1.0, Conan introduces new settings to model the GNU convention triplet:

- **build platform settings**:
  - **os_build**: Operating system of the build system.
• **arch_build**: Architecture system of the build system.

These settings are detected the first time you run Conan with the same values than the host settings, so by default, we are doing **native building**. You will probably never need to change the value of this setting because they describe where are you running Conan.

**host** platform settings:
• **os**: Operating system of the host system.
• **arch**: Architecture of the host system.
• **compiler**: Compiler of the host system (to declare compatibility of libs in the host platform)
• … (all the regular settings)

These settings are the regular Conan settings; already present before supporting the GNU triplet convention. If you are cross-building, you have to change them according to the host platform.

**target** platform:
• **os_target**: Operating system of the target system.
• **arch_target**: Architecture of the target system.

If you are building a compiler, these settings specify where the compiled code will run.

### 10.1.3 Cross-building with Conan

If you want to cross-build a Conan package (for example on your Linux machine) to build the zlib Conan package for Windows, you need to tell Conan where to find your cross-compiler/toolchain.

There are two approaches:

- Install the toolchain in your computer and use a profile to declare the settings and point to the needed tools/libraries in the toolchain using the [env] section to declare, at least, the `CC` and `CXX` environment variables.
- Package the toolchain as a Conan package and include it as a build_require.

**Using profiles**

Create a profile with:

- A **[settings]** section containing the needed settings: `os_build`, `arch_build` and the regular settings `os`, `arch`, `compiler`, `build_type` and so on.
- An [env] section with a PATH variable pointing to your installed toolchain. Also any other variable that the toolchain expects (read the docs of your compiler). Some build systems need a variable `SYSROOT` to locate where the host system libraries and tools are.

**Linux to Windows**

- Install the needed toolchain, in Ubuntu:
  
  ```bash
  sudo apt-get install g++-mingw-w64 gcc-mingw-w64
  ```
- Create a file named `linux_to_win64` with the contents:
toolchain=/usr/x86_64-w64-mingw32 # Adjust this path
target_host=x86_64-w64-mingw32
cc_compiler=gcc
cxx_compiler=g++

[env]
CONAN_CMAKE_FIND_ROOT_PATH=$toolchain
CHOST=$target_host
AR=$target_host-ar
AS=$target_host-as
RANLIB=$target_host-ranlib
CC=$target_host-$cc_compiler
CXX=$target_host-$cxx_compiler
STRIP=$target_host-strip
RC=$target_host-windres

[settings]
# We are building in Ubuntu Linux
os_build=Linux
arch_build=x86_64

# We are cross-building to Windows
os=Windows
arch=x86_64
compiler=gcc

# Adjust to the gcc version of your MinGW package
compiler.version=7.3
compiler.libcxx=libstdc++11
build_type=Release

- Clone an example recipe or use your own recipe:

  git clone https://github.com/memsharded/conan-hello.git

- Call conan create using the created linux_to_win64

  $ cd conan-hello && conan create . conan/testing --profile ../linux_to_win64

  ...[ 50%] Building CXX object CMakeFiles/example.dir/example.cpp.obj
  [100%] Linking CXX executable bin/example.exe
  [100%] Built target example

A bin/example.exe for Win64 platform has been built.

**Windows to Raspberry Pi (Linux/ARM)**

- Install the toolchain: http://gnutoolchains.com/raspberry/ You can choose different versions of the GCC cross compiler. Choose one and adjust the following settings in the profile accordingly.

- Create a file named win_to_rpi with the contents:

  target_host=arm-linux-gnueabihf
  standalone_toolchain=C:/sysgcc/raspberry
  cc_compiler=gcc
cxx_compiler=g++

(continues on next page)
The profiles to target Linux are all very similar. You probably just need to adjust the variables declared at the top of the profile:

- **target_host**: All the executables in the toolchain starts with this prefix.
- **standalone_toolchain**: Path to the toolchain installation.
- **cc_compiler/cxx_compiler**: In this case gcc/g++, but could be clang/clang++.
- Clone an example recipe or use your own recipe:

  ```
git clone https://github.com/memsharded/conan-hello.git
  ```

- Call **conan create** using the created profile.

  ```
$ cd conan-hello && conan create . conan/testing --profile=../win_to_rpi
...[ 50%] Building CXX object CMakeFiles/example.dir/example.cpp.obj
[100%] Linking CXX executable bin/example
[100%] Built target example
  ```

A **bin/example** for Raspberry Pi (Linux/armv7hf) platform has been built.

**Windows to Windows CE**

The Windows CE (WinCE) operating system is supported for CMake and MSBuild. Since WinCE depends on the MSVC compiler, Visual Studio and the according Windows CE platform SDK for the WinCE device have to be installed on the build host.

The **os.platform** defines the WinCE Platform SDK and is equal to the Platform in Visual Studio.

Some examples for Windows CE platforms:

- **SDK_AM335X_SK_WEC2013_V310**
The `os.version` defines the WinCE version and must be "5.0", "6.0" or "7.0".

CMake supports Visual Studio 2008 (`compiler.version=9`) and Visual Studio 2012 (`compiler.version=11`).

Example of an Windows CE conan profile:

```plaintext
[settings]
os=WindowsCE
os.version=8.0
os.platform=Toradex_CE800 (ARMV7)
arch=armv7
compiler=Visual Studio
compiler.version=11

# Release configuration
build_type=Release
compiler.runtime=MD
```

**Note:** Further information about CMake and WinCE can be found in the CMake documentation:

CMake - Cross Compiling for Windows CE

---

**Linux/Windows/macOS to Android**

Cross-building a library for Android is very similar to the previous examples, except the complexity of managing different architectures (armeabi, armeabi-v7a, x86, arm64-v8a) and the Android API levels.

Download the Android NDK [here](#) and unzip it.

**Note:** If you are in Windows the process will be almost the same, but unzip the file in the root folder of your hard disk (C:\) to avoid issues with path lengths.

Now you have to build a standalone toolchain. We are going to target the “arm” architecture and the Android API level 21. Change the `--install-dir` to any other place that works for you:

```bash
$ cd build/tools
$ python make_standalone_toolchain.py --arch=arm --api=21 --stl=libc++ --install-dir=/myfolder/arm_21_toolchain
```

**Note:** You can generate the standalone toolchain with several different options to target different architectures, API levels etc.

Check the Android docs: standalone toolchain

To use the clang compiler, create a profile `android_21_arm_clang`. Once again, the profile is very similar to the RPI one:
standalone_toolchain=/myfolder/arm_21_toolchain # Adjust this path
cc_compiler=clang
cxx_compiler=clang++

[settings]
compiler=clang
compiler.version=5.0
compiler.libcxx=libc++
os=Android
os.api_level=21
arch=armv7
build_type=Release

[env]
CONAN_CMAKE_FIND_ROOT_PATH=$standalone_toolchain/sysroot
PATH=[$standalone_toolchain/bin]
CHOST=$target_host
AR=$target_host-ar
AS=$target_host-as
RANLIB=$target_host-ranlib
CC=$target_host-$cc_compiler
CXX=$target_host-$cxx_compiler
LD=$target_host-ld
STRIP=$target_host-strip
CFLAGS= -fPIE -fPIC -I$standalone_toolchain/include/c++/4.9.x
CXXFLAGS= -fPIE -fPIC -I$standalone_toolchain/include/c++/4.9.x
LDFLAGS= -pie

You could also use gcc using this profile arm_21_toolchain_gcc, changing the cc_compiler and
cxx_compiler variables, removing -fPIE flag and, of course, changing the [settings] to match the gcc
toolchain compiler:

standalone_toolchain=/myfolder/arm_21_toolchain
target_host=arm-linux-androideabi
cc_compiler=gcc
cxx_compiler=g++

[settings]
compiler=gcc
compiler.version=4.9
compiler.libcxx=libstdc++
os=Android
os.api_level=21
arch=armv7
build_type=Release

[env]
CONAN_CMAKE_FIND_ROOT_PATH=$standalone_toolchain/sysroot
PATH=[$standalone_toolchain/bin]
CHOST=$target_host
AR=$target_host-ar
AS=$target_host-as
RANLIB=$target_host-ranlib
CC=$target_host-$cc_compiler
CXX=$target_host-$cxx_compiler
LD=$target_host-ld

(continues on next page)
STRIP=$target_host-strip
CFLAGS= -fPIC -I$standalone_toolchain/include/c++/4.9.x
CXXFLAGS= -fPIC -I$standalone_toolchain/include/c++/4.9.x
LDFLAGS=

• Clone, for example, the zlib library to try to build it to Android

    git clone https://github.com/conan-community/conan-zlib.git

• Call `conan create` using the created profile.

    $ cd conan-zlib && conan create . conan/testing --profile=../android_21_arm_clang

...  
-- Build files have been written to: /tmp/conan-zlib/test_package/build/  
→  
- ba0b9dbae0576b9a23ce7005180b0e4fdef1198  
Scanning dependencies of target enough  
[ 50%] Building C object CMakeFiles/enough.dir/enough.c.o  
[100%] Linking C executable bin/enough  
[100%] Built target enough  
zlib/1.2.11@conan/testing (test package): Running test()

A `bin/enough` for Android ARM platform has been built.

**Using build requires**

Instead of manually downloading the toolchain and creating a profile, you can create a Conan package with it. The toolchain Conan package needs to fill the `env_info` object in the `package_info()` method with the same variables we’ve specified in the examples above in the `[env]` section of profiles.

A layout of a Conan package for a toolchain could looks like this:

```python
from conans import ConanFile  
import os

class MyToolchainXXXConan(ConanFile):
    name = "my_toolchain"  
    version = "0.1"  
    settings = "os_build", "arch_build"

    def build(self):
        # Typically download the toolchain for the 'build' host
        url = "http://fake_url.com/installers/%s/%s/toolchain.tgz" % (os_build, os_arch)
        tools.download(url, "toolchain.tgz")  
        tools.unzip("toolchain.tgz")

    def package(self):
        # Copy all the
        self.copy("*", dst="", src="toolchain")

    def package_info(self):
        bin_folder = os.path.join(self.package_folder, "bin")
        self.env_info.path.append(bin_folder)
        self.env_info.CC = os.path.join(bin_folder, "mycompiler-cc")
```

(continues on next page)
Finally, when you want to cross-build a library, the profile to be used will include a [build_requires] section with the reference to our new packaged toolchain. This will also contain a [settings] section with the same settings from the examples above.

**Example: Darwin Toolchain**

Check the Darwin Toolchain package in conan-center. You can use a profile like the following to cross-build your packages for iOS, watchOS and tvOS:

```
include(default)

[settings]
os=iOS
os.version=9.0
arch=armv7

[build_requires]
darwin-toolchain/1.0@theodelrieu/stable
```

$ conan install . --profile ios_profile

See also:

- Check the *Creating conan packages to install dev tools* to learn more about how to create Conan packages for tools.
- Check the mingw-installer build require recipe as an example of packaging a compiler.

### Using Docker images

You can use some available Docker images with Conan preinstalled images to cross-build Conan packages. Currently there are i386, armv7 and armv7hf images with the needed packages and toolchains installed to cross-build.

**Example:** Cross-building and uploading a package along with all its missing dependencies for Linux/armv7hf is done in few steps:

```
$ git clone https://github.com/conan-community/conan-openssl
$ cd conan-openssl
$ docker run -it -v $(pwd):/home/conan/project --rm conanio/gcc49-armv7hf /bin/bash

# Now we are running on the conangcc49-armv7hf container
$ sudo pip install conan --upgrade
$ cd project

$ conan create . user/channel --build missing
$ conan remote add myremoteARMV7 http://some.remote.url
$ conan upload '*' -r myremoteARMV7 --all
```

Check the section: *How to run Conan with Docker* to know more.
Preparing recipes to be cross-compiled

If you use the build helpers `AutoToolsBuildEnvironment` or `CMake`, Conan will adjust the configuration accordingly to the specified settings.

If don’t, you can always check the `self.settings.os`, `self.settings.build_os`, `self.settings.arch` and `self.settings.build_arch` settings values and inject the needed flags to your build system script.

You can use this tool to check if you are cross-building:

- `tools.cross_building(self.settings)` (returns True or False)

10.1.4 ARM architecture reference

Remember that the Conan settings are intended to unify the different names for operating systems, compilers, architectures etc.

Conan has different architecture settings for ARM: `armv6`, `armv7`, `armv7hf`, `armv8`. The “problem” with ARM architecture is that it’s frequently named in different ways, so maybe you are wondering what setting do you need to specify in your case.

Here is a table with some typical ARM platforms:

<table>
<thead>
<tr>
<th>Platform</th>
<th>Conan setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Raspberry PI 1</td>
<td><code>armv6</code></td>
</tr>
<tr>
<td>Raspberry PI 2</td>
<td><code>armv7</code> or <code>armv7hf</code> if we want to use the float point hard support</td>
</tr>
<tr>
<td>Raspberry PI 3</td>
<td><code>armv6</code> also known as <code>armv64-v8a</code></td>
</tr>
<tr>
<td>Visual Studio</td>
<td><code>armv7</code> currently Visual Studio builds <code>armv7</code> binaries when you select ARM.</td>
</tr>
<tr>
<td>Android armbeabi-v7a</td>
<td><code>armv7</code></td>
</tr>
<tr>
<td>Android armv64-v8a</td>
<td><code>armv8</code></td>
</tr>
<tr>
<td>Android armeabi</td>
<td><code>armv6</code> (as a minimal compatible, will be compatible with v7 too)</td>
</tr>
</tbody>
</table>

See also:

Reference links

**ARM**


**ANDROID**


**VISUAL STUDIO**


See also:

- See `conan.conf` file and `Environment variables` sections to know more.
- See `AutoToolsBuildEnvironment` build helper reference.
- See `CMake build helper` reference.
- See `CMake` cross-building wiki to know more about cross-building with `CMake`. 

10.1. Cross-building
10.2 Windows Subsystems

On Windows, you can run different subsystems that enhance the operating system with UNIX capabilities. Conan supports MSYS2, CYGWIN, WSL and in general any subsystem that is able to run a bash shell.

Many libraries use these subsystems in order to use the Unix tools like the Autoconf suite that generates Makefiles.

The difference between MSYS2 and CYGWIN is that MSYS2 is oriented to the development of native Windows packages, while CYGWIN tries to provide a complete POSIX-like system to run any Unix application on it.

For that reason, we recommend the use of MSYS2 as a subsystem to be used with Conan.

10.2.1 Operation Modes

The MSYS2 and CYGWIN can be used with different operation modes:

- You can use them together with MinGW to build Windows-native software.
- You can use them together with any other compiler to build Windows-native software, even with Visual Studio.
- You can use them with MinGW to build specific software for the subsystem, with a dependency to a runtime DLL (msys-2.0.dll and cygwin1.dll)

If you are building specific software for the subsystem, you have to specify a value for the setting `os.subsystem`, if you are only using the subsystem for taking benefit of the UNIX tools but generating native Windows software, you shouldn’t specify it.

10.2.2 Running commands inside the subsystem

`self.run()`

In a Conan recipe, you can use the `self.run` method specifying the parameter `win_bash=True` that will call automatically to the tool `tools.run_in_windows_bash`.

It will use the `bash` in the path or the `bash` specified for the environment variable `CONAN_BASH_PATH` to run the specified command.

Conan will automatically escape the command to match the detected subsystem. If you also specify the `msys_mingw` parameter to False, and the subsystem is MSYS2 it will run in Windows-native mode, the compiler won’t link against the msys-2.0.dll.

**AutoToolsBuildEnvironment**

In the constructor of the build helper, you have the `win_bash` parameter. Set it to `True` to run the `configure` and `make` commands inside a bash.

10.2.3 Controlling the build environment

Building software in a Windows subsystem for a different compiler than MinGW can sometimes be painful. The reason is how the subsystem finds your compiler/tools in your system.
For example, the icu library requires Visual Studio to be built in Windows, but also a subsystem able to build the Makefile. A very common problem and example of the pain is the link.exe program. In the Visual Studio suite, link.exe is the linker, but in the MSYS2 environment the link.exe is a tool to manage symbolic links.

Conan is able to prioritize the tools when you use build_requires, and put the tools in the PATH in the right order.

There are some packages you can use as build_requires:

- From Conan-center:
  - mingw_installer/1.0@conan/stable: MinGW compiler installer as a Conan package.
  - msys2_installer/latest@bincrafters/stable: MSYS2 subsystem as a Conan package.
  - cygwin_installer/2.9.0@bincrafters/stable: Cygwin subsystem as a Conan package.

For example, create a profile and name it msys2_mingw with the following contents:

```makefile
[build_requires]
mingw_installer/1.0@conan/stable
msys2_installer/latest@bincrafters/stable

[settings]
os_build=Windows
os=Windows
arch=x86_64
arch_build=x86_64
compiler=gcc
compiler.version=4.9
compiler.exception=seh
compiler.libcxx=libstdc++11
compiler.threads=posix
build_type=Release
```

Then you can have a conanfile.py that can use self.run() with win_bash=True to run any command in a bash terminal or use the AutoToolsBuildEnvironment to invoke configure/make in the subsystem:

```python
from conans import ConanFile
import os

class MyToolchainXXXConan(ConanFile):
    name = "mylib"
    version = "0.1"
    ...

    def build(self):
        self.run("some_command", win_bash=True)

        env_build = AutoToolsBuildEnvironment(self, win_bash=True)
        env_build.configure()
        env_build.make()
        ...
```

Apply the profile in your recipe to create a package using the MSYS2 and MINGW:

```
$ conan create . user/testing --profile msys2_mingw
```

10.2. Windows Subsystems
As we included in the profile the MinGW and then the MSYS2 build require, when we run a command, the PATH will contain first the MinGW tools and finally the MSYS2.

What could we do with the Visual Studio issue with link.exe? You can pass an additional parameter to run_in_windows_bash with a dictionary of environment variables to have more priority than the others:

```python
def build(self):
    # ...
    vs_path = tools.vcvars_dict(self.settings)['PATH']  # Extract the path from the vcvars_dict tool
    tools.run_in_windows_bash(self, command, env={'PATH': vs_path})
```

So you will get first the link.exe from the Visual Studio.

Also, Conan has a tool tools.remove_from_path that you can use in a recipe to temporarily remove a tool from the path if you know that it can interfere with your build script:

```python
class MyToolchainXXXConan(ConanFile):
    name = "mylib"
    version = "0.1"
    ...

    def build(self):
        with tools.remove_from_path("link"):
            # Call something
            self.run("some_command", win_bash=True)
```

```python```
This section provides an introduction to extension capabilities of Conan:

### 11.1 Customizing settings

There is a file in `<userhome>/conan/settings.yml` that contains a default definition of the allowed settings values for Conan package recipes. It looks like:

```
os:
  Windows:
    subsystem: [None, cygwin, msys, msys2, wsl]
  Linux:
  Macos:
    version: [None, "10.6", "10.7", "10.8", "10.9", "10.10", "10.11", "10.12",
      "10.13", "10.14"]
  Android:
    api_level: ANY
  iOS:
      "10.0", "10.1", "10.2", "10.3", "11.0", "11.1", "11.2", "11.3", "11.4", "12.0",
      "12.1"]
  watchOS:
    version: ["4.0", "4.1", "4.2", "4.3", "5.0", "5.1"]
  FreeBSD:
  SunOS:
  Emscripten:
arch: [x86, x86_64, ppc32, ppc64le, ppc64, armv4, armv4i, armv5te, armv6,
  armv7, armv7hf, armv7k, armv8, armv8_32, armv8.3, sparc, sparcv9, mips,
  mips64, avr, s390, s390x, asm.js, wasm]

compiler:
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",
    "7", "7.1", "7.2", "7.3",
    "8", "8.1", "8.2",
    "9"]
libcxx: [libstdc++, libstdc++11]

tthreads: [None, posix, win32] # Windows MinGW
exception: [None, dwarf2, sjlj, seh] # Windows MinGW
cppstd: [None, 98, gnu98, 11, gnu11, 14, gnu14, 17, gnu17, 20, gnu20]
Visual Studio:
  runtime: [MD, MT, MTd, MDD]
```
version: ['8', '9', '10', '11', '12', '14', '15', '16']
cppstd: [None, 14, 17, 20]

This are the default settings and values. They are a common syntax and notation for having package binary IDs that are common to all developers. They are also used for validation, for example if you write in a profile [settings] something like `os=Windos` (note the typo), then it will raise an error, telling you it is not a recognized `os` and offering a list of available `os`. Also, note how the sub-settings are different for different platforms, for example the standard C++ library (`compiler.libcxx`) exists for the gcc compiler, but not for Visual Studio compiler. And in the same way, Visual Studio has a runtime sub-setting that is missing in gcc. Trying to incorrectly use or define these sub-settings in the wrong compiler will also raise an error.

These settings are good for defining a base for Open Source packages, and for a large number of mainstream configurations. But it is likely that you might need finer detail of definition of the binaries that are being created.

For example, it is possible that you are managing binaries for older Linux distros, like RHEL 6, or old Centos, besides other modern distributions. The problem is that the binaries compiled for modern distributions will not work (will not be binary compatible, or ABI incompatible) in those older distributions, mainly because of different versions of glibc. We would need a way to model the differences of the binaries for those platforms.

### 11.1.1 Adding new settings

It is possible to add new settings at the root of the `settings.yml` file, something like:

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

```python
class Pkg(ConanFile):
  settings = "os", "compiler", "build_type", "arch", "distro"
```

The value `None` allows for not defining it (which would be a default value, valid for all other distros). It is possible to define values for it in the profiles:

```yaml
[settings]
os = "Linux"
distro = "CentOS"
compiler = "gcc"
```

And use their values to affect our build if desired:

```python
class Pkg(ConanFile):
  settings = "os", "compiler", "build_type", "arch", "distro"
  def build(self):
    cmake = CMake(self)
    if self.settings.distro == "CentOS":
```

(continues on next page)
11.1.2 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 possible, it might make more sense for other cases, but for this example it is more adequate to add it as above subsetting of the Linux OS:

```
<table>
<thead>
<tr>
<th>os:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows:</td>
</tr>
<tr>
<td>subsystem: [None, cygwin, msys, msys2, wsl]</td>
</tr>
<tr>
<td>Linux:</td>
</tr>
<tr>
<td>distro: [None, RHEL6, CentOS, Debian]</td>
</tr>
</tbody>
</table>
```

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 subsetting, it will be automatically taken into account in all recipes that declare an `os` setting. Note that having a value of `distro=None` 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 subsetting, won’t be usable anymore.

The sub-setting can also be accessed from recipes:

```python
class Pkg(ConanFile):
    settings = "os", "compiler", "build_type", "arch" # Note, no "distro" defined.

    def build(self):
        cmake = CMake(self)
        if self.settings.os == "Linux" and self.settings.os.distro == "CentOS":
            cmake.definitions["SOME_CENTOS_FLAG"] = "Some CentOS Value"
```

11.1.3 Add new values

In the same way we have added a new distro subsetting, it is possible to add new values to existing settings and subsettings. 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:

```
<table>
<thead>
<tr>
<th>os:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows:</td>
</tr>
<tr>
<td>subsystem: [None, cygwin, msys, msys2, wsl]</td>
</tr>
</tbody>
</table>
...```
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 built-in 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 Visual Studio setting and sub-settings, but not the new compiler. For those cases, custom logic can be implemented in the recipes:

```python
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 . %s" % my_custom_compile)
```

**Note:** You can also remove items from `settings.yml` file. You can remove 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.

## 11.2 Python requires: reusing code [EXPERIMENTAL]

**Warning:** This is an experimental feature subject to breaking changes in future releases.

The `python_requires()` feature is a very convenient way to share files and code between different recipes. A `Python Requires` is just like any other recipe, it is the way it is required from the consumer what makes the difference.

The `Python Requires` recipe file, besides exporting its own required sources, can export files to be used by the consumer recipes and also python code in the recipe file itself.

Let's have a look at an example showing all its capabilities (you can find all the sources in Conan examples repository):

- Python requires recipe:

```
import os
import shutil
from conans import ConanFile, CMake, tools
from scm_utils import get_version
```
class PythonRequires(ConanFile):
    name = "pyreq"
    version = "version"

    exports = "scm_utils.py"
    exports_sources = "CMakeLists.txt"

def get_conanfile():

class BaseConanFile(ConanFile):

    settings = "os", "compiler", "build_type", "arch"
    options = {"shared": [True, False]}
    default_options = {"shared": False}
    generators = "cmake"
    exports_sources = "src/*"

    def source(self):
        # Copy the CMakeLists.txt file exported with the python
        # requires
        pyreq = self.python_requires["pyreq"]
        shutil.copy(src=os.path.join(pyreq.exports_sources_folder,
                                     "CMakeLists.txt"),
                     dst=self.source_folder)

        # Rename the project to match the consumer name
        tools.replace_in_file(os.path.join(self.source_folder,
                                            "CMakeLists.txt"),
                               "add_library(mylibrary ${sources})",
                               "add_library({} ${{sources}})".
                               format(self.name))

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

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

    def package_info(self):
        self.cpp_info.libs = [self.name]

    return BaseConanFile

• Consumer recipe

    from conans import ConanFile, python_requires

(continues on next page)
We must make available for other to use the recipe with the Python Requires, this recipe won’t have any associated binaries, only the sources will be needed, so we only need to execute the export and upload commands:

```
$ conan export . pyreq/version@user/channel
$ conan upload pyreq/version@user/channel -r=myremote
```

Now any consumer will be able to reuse the business logic and files available in the recipe, let’s have a look at the most common use cases.

### 11.2.1 Import a python requires

To import a recipe as a Python requires it is needed to call the `python_requires()` function with the reference as the only parameter:

```
base = python_requires("pyreq/version@user/channel")
```

All the code available in the `conanfile.py` file of the imported recipe will be available in the consumer through the `base` variable.

**Important:** There are several important considerations regarding `python_requires()`:

- They are required at every step of the conan commands. If you are creating a package that `python_requires("MyBase/...")`, the `MyBase` package should be already available in the local cache or to be downloaded from the remotes. Otherwise, conan will raise a “missing package” error.
- They do not affect the package binary ID (hash). Depending on different version, or different channel of such `python_requires()` do not change the package IDs as the normal dependencies do.
- They are imported only once. The python code that is reused is imported only once, the first time it is required. Subsequent requirements of that conan recipe will reuse the previously imported module. Global initialization at parsing time and global state are discouraged.
- They are transitive. One recipe using `python_requires()` can be also consumed with a `python_requires()` from another package recipe.
- They are not automatically updated with the `--update` argument from remotes.
- Different packages can require different versions in their `python_requires()`. They are private to each recipe, so they do not conflict with each other, but it is the responsibility of the user to keep consistency.
- They are not overridden from downstream consumers. Again, as they are private, they are not affected by other packages, even consumers.
11.2.2 Reuse python sources

In the example proposed we are using two functions through the `base` variable: `base.get_conanfile()` and `base.get_version()`. The first one is defined directly in the `conanfile.py` file, but the second one is in a different source file that was exported together with the `pyreq/version@user/channel` recipe using the `exports` attribute.

This works without any Conan magic, it is just plain Python and you can even return a class from a function and inherit from it. That’s just what we are proposing in this example: all the business logic is contained in the Python Requires so every recipe will reuse it automatically. The consumer only needs to define the `name` and `version`:

```python
from conans import ConanFile, python_requires

base = python_requires("pyreq/version@user/channel")

class ConsumerConan(base.get_conanfile()):
    name = "consumer"
    version = "version"

    # Everything else is inherited
```

while all the functional code is defined in the `python requires` recipe file:

```python
from conans import ConanFile, python_requires

[...]
def get_conanfile():
    class BaseConanFile(ConanFile):
        def source(self):
            [...]
        def build(self):
            [...]
```

11.2.3 Reuse source files

Up to now, we have been reusing python code, but we can also package files within the `python requires` recipe and consume them afterward, that’s what we are doing with a `CMakeLists.txt` file, it will allow us to share the CMake code and ensure that all the libraries using the same `python requires` will have the same build script. These are the relevant code snippets from the example files:

- The `python requires` exports the needed sources (the file exists next to this `conanfile.py`):

  ```python
class PythonRequires(ConanFile):
    name = "pyreq"
    version = "version"

    exports_sources = "CMakeLists.txt"

    [...]
```

The file will be exported together with the recipe `pyreq/version@user/channel` during the call to `conan export . pyreq/version@user/channel` as it is expected for any Conan package.
• The consumer recipe will copy the file from the python requires folder, we need to make this copy ourselves, there is nothing run automatically during the python_requires() call:

```python
class BaseConanFile(ConanFile):
    
    def source(self):
        # Copy the CMakeLists.txt file exported with the python requires
        pyreq = self.python_requires["pyreq"]
        shutil.copy(src=os.path.join(pyreq.exports_sources_folder,
                                   "CMakeLists.txt"),
                   dst=self.source_folder)

        # Rename the project to match the consumer name
        tools.replace_in_file(os.path.join(self.source_folder,
                                   "CMakeLists.txt"),
                               "add_library(mylibrary ${sources})",
                               "add_library({} ${{sources}})".format(self.
                               name))
```

As you can see, in the inherited source() method, we are copying the CMakeLists.txt file from the exports_sources folder of the python requires (take a look at the python_requires attribute), and modifying a line to name the library with the current recipe name.

In the example, our ConsumerConan class will also inherit the build(), package() and package_info() method, turning the actual conanfile.py of the library into a mere declaration of the name and version.

You can find the full example in the Conan examples repository.

### 11.3 Hooks

**Warning:** This is an experimental feature subject to breaking changes in future releases.

The Conan hooks is a feature intended to extend the Conan functionalities and let users customize the client behavior at determined points.

#### 11.3.1 Hook structure

Hooks are Python files containing pre and post functions that will be executed before and after a particular task performed by the Conan client. Those tasks could be Conan commands, recipe interactions such as exporting or packaging, or interactions with the remotes.

Here is an example of a simple hook:

```python
from conans import tools

def pre_export(output, conanfile, conanfile_path, reference, **kwargs):
    test = "$%/s\%s" % (reference.name, reference.version)
    for field in ["url", "license", "description"]:
```

(continues on next page)
This hook checks the recipe content prior to it being exported and prior to downloading the sources. Basically the \texttt{pre\_export()} function checks the attributes of the \texttt{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 \texttt{output}. This is done \textbf{before} the recipe is exported to the local cache.

The \texttt{pre\_source()} function checks if the recipe contains a \texttt{source()} method (this time it is using the \texttt{conanfile} content instead of the \texttt{conanfile} object) and in that case it checks if the download of the sources are likely coming from immutable places (a compressed file or a determined \texttt{git checkout}). This is done \textbf{before} the \texttt{source()} method of the recipe is called.

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.

Each function receives some parameters but not all of them are available for all functions as this may change depending on the context of the commands being executed, such as the recipe being in the local cache or not.

**Important:** A detailed description of the functions allowed and its parameters as well as their execution can be found in it dedicated reference section: \textit{Hooks [EXPERIMENTAL]}.

Other useful task where a hook may come handy are the upload and download actions. There are \texttt{pre} and \texttt{post} functions for every download/upload as a whole and for fine download tasks such as recipe and package downloads/uploads.

For example they can be used to sign the packages (including a file with the signature) when the package is created and check that signature every time they are downloaded.
Listing 2: `signing_hook.py`

```python
import os
from conans import tools

SIGNATURE = "this is my signature"

def post_package(output, conanfile, conanfile_path, **kwargs):
    sign_path = os.path.join(conanfile.package_folder, ".sign")
    tools.save(sign_path, SIGNATURE)
    output.success("Package signed successfully")

def post_download_package(output, conanfile_path, reference, package_id, remote_name,
                          **kwargs):
    package_path = os.path.abspath(os.path.join(os.path.dirname(conanfile_path), "..",
                          "package", package_id))
    sign_path = os.path.join(package_path, ".sign")
    content = tools.load(sign_path)
    if content != SIGNATURE:
        raise Exception("Wrong signature")
```

### 11.3.2 Importing from a module

The hook interface should always be placed inside a Python file with the name of the hook and stored in the `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 3: `example_hook.py`

```python
import requests
from conans import tools

def post_export(output, conanfile, conanfile_path, reference, **kwargs):
    cmakelists_path = os.path.join(os.path.dirname(conanfile_path), "CMakeLists.txt")
    tools.replace_in_file(cmakelists_path, "PROJECT(MyProject)", "PROJECT(MyProject)
    "CPP")
    r = requests.get('https://api.github.com/events')
```

You can also import functionalities from a relative module:

```
hooks
| custom_module
| custom.py
| __init__.py
| my_hook.py
```

Inside the `custom.py` from `my custom_module` there is:

```python
def my_printer(output):
    output.info("my_printer(): CUSTOM MODULE")
```

And it can be used in hook importing the module:

```python
from custom_module.custom import my_printer
```

(continues on next page)
def pre_export(output, conanfile, conanfile_path, reference, **kwargs):
    my_printer(output)

11.3.3 Storage, activation and sharing

Hooks are Python files stored under ~/.conan/hooks folder and their file name should be the same used for activation (the .py extension could be indicated or not).

The activation of the hooks is done in the conan.conf section named [hooks]. The hook names or paths listed under this section will be considered activated.

Listing 4: conan.conf

...  
[hooks]  
attribute_checker.py  
conan-center.py  
my_custom_hook/hook.py

They can be easily activated and deactivated from the command line using the conan config set command:

$ conan config set hooks.my_custom_hook/hook  # Activates 'my_custom_hook'
$ conan config rm hooks.my_custom_hook/hook   # Deactivates 'my_custom_hook'

There is also an environment variable CONAN_HOOKS to list the active hooks. Hooks listed in conan.conf will be loaded into this variable and values in the environment variable will be used to load the hooks.

Hooks are considered part of the Conan client configuration and can be shared as usual with the conan config install command. However, they can also be managed in isolated git repositories cloned into the hooks folder:

$ cd ~/.conan/hooks  
$ git clone https://github.com/conan-io/hooks.git conan_hooks  
$ conan config set hooks.conan_hooks/hooks/conan-center.py

This way you can easily change from one version to another.

11.3.4 Official Hooks

There is a simple attribute_checker hook ready to be used in Conan. You can take it as a starting point to create your own ones.

attribute_checker

This hook is shipped together with the Conan client and it warns the user when recipes do not contain some metadata attributes.

Listing 5: attribute_checker.py

def pre_export(output, conanfile, conanfile_path, reference, **kwargs):
    # Check basic meta-data
    for field in ["url", "license", "description"]:
field_value = getattr(conanfile, field, None)
if not field_value:
    output.warn("Conanfile doesn't have \'%s\'. It is recommended to add it as an attribute" % field)

This hook comes activated by default.
12.1 Build systems

Conan can be integrated with any build system. This can be done with:

- **Generators**: Conan can write file/s in different formats gathering all the information from the dependency tree, like include directories, library names, library dirs. . .

- **Build Helpers**: Conan provides some classes to help calling your build system, translating the settings and options to the arguments, flags or environment variables that your build system expect.

12.1.1 CMake

Conan can be integrated with CMake using generators, build helpers and custom `findXXX.cmake` files:

**cmake generator**

If you are using CMake to build your project, you can use the `cmake` generator to define all your requirements in CMake syntax. It creates a file named `conanbuildinfo.cmake` that can be imported from your `CMakeLists.txt`.

```
[generators]
cmake
```

Listing 1: `conanfile.txt`

When `conan install` is executed, a file named `conanbuildinfo.cmake` is created.

You can include `conanbuildinfo.cmake` in your project’s `CMakeLists.txt` to manage your requirements. The inclusion of `conanbuildinfo.cmake` doesn’t alter the CMake environment at all. It simply provides `CONAN_` variables and some useful macros.
Global variables approach

The simplest way to consume it would be to invoke the `conan_basic_setup()` macro, which will basically set global include directories, libraries directories, definitions, etc. so typically it is enough to call:

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

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

The `conan_basic_setup()` is divided into smaller macros that should be self explanatory. If you need to do something different, you can just call them individually.

**Note:** This approach makes all dependencies visible to all CMake targets and may also increase the build times due to unneeded include and library path components. This is particularly relevant if you have multiple targets with different dependencies. In that case, you should consider using the *Targets approach*.

Targets approach

For **modern cmake (>=3.1.2)**, you can use the following approach:

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

add_executable(timer timer.cpp)
target_link_libraries(timer CONAN_PKG::Poco)
```

Using `TARGETS` as argument, `conan_basic_setup()` will internally call the macro `conan_define_targets()` which defines cmake INTERFACE IMPORTED targets, one per package. These targets, named `CONAN_PKG::PackageName` can be used to link against, instead of using global cmake setup.

**See also:**
Check the *CMake generator* section to read more.

**Note:** The `CMAKE_MODULE_PATH` and `CMAKE_PREFIX_PATH` contain the paths to the `self.info. builddirs` of every required package. By default, the root package folder is the only one declared in `builddirs`. Check `cpp_info` for more information.

cmake_multi generator

cmake_multi generator is intended for CMake multi-configuration environments, like Visual Studio and Xcode IDEs that do not configure for a specific `build_type`, like Debug or Release, but rather can be used for both and switch among Debug and Release configurations with a combo box or similar control. The project configuration for cmake is different, in multi-configuration environments, the flow would be:

```
$ cmake .. --G "Visual Studio 14 Win64"
# Now open the IDE (.sln file) or
$ cmake --build . --config Release
```
While in single-configuration environments (Unix Makefiles, etc):

```bash
$ cmake .. -G "Unix Makefiles" -DCMAKE_BUILD_TYPE=Release
# Build from your IDE, launching make, or
$ cmake --build .
```

The `CMAKE_BUILD_TYPE` default, if not specified is `Debug`.

With the regular conan `cmake` generator, only 1 configuration at a time can be managed. Then, it is a universal, homogeneous solution for all environments. This is the recommended way, using the regular `cmake` generator, and just go to the command line and switch among configurations:

```bash
$ conan install . -s build_type=Release ...
# Work in release, then, to switch to Debug dependencies
$ conan install . -s build_type=Debug ...
```

However, end consumers with heavy usage of the IDE, might want a multi-configuration build. The `cmake_multi` experimental generator is able to do that. First, both Debug and Release dependencies have to be installed:

```bash
$ conan install . -g cmake_multi -s build_type=Release ...
$ conan install . -g cmake_multi -s build_type=Debug ...
```

These commands will generate 3 files: `conanbuildinfo_release.cmake`, `conanbuildinfo_debug.cmake`, and `conanbuildinfo_multi.cmake`, which includes the other two, and enables its use.

**Warning:** The `cmake_multi` generator is designed as a helper for consumers, but not for creating packages. If you also want to create a package, see `Creating packages` section.

**Global variables approach**

The consumer project might write a `CMakeLists.txt` like:

```cmake
project(MyHello)
cmake_minimum_required(VERSION 2.8.12)
include(${CMAKE_BINARY_DIR}/conanbuildinfo_multi.cmake)
conan_basic_setup()

add_executable(say_hello main.cpp)
foreach(_LIB ${CONAN_LIBS_RELEASE})
  target_link_libraries(say_hello optimized ${_LIB})
endforeach()
foreach(_LIB ${CONAN_LIBS_DEBUG})
  target_link_libraries(say_hello debug ${_LIB})
endforeach()
```

**Targets approach**

Or, if using the modern `cmake` syntax with targets (where `Hello1` is an example package name that the executable `say_hello` depends on):

```
12.1. Build systems
```
project(MyHello)
cmake_minimum_required(VERSION 2.8.12)
include(${CMAKE_BINARY_DIR}/conanbuildinfo_multi.cmake)
conan_basic_setup(TARGETS)
add_executable(say_hello main.cpp)
target_link_libraries(say_hello CONAN_PKG::Hello1)

There's also a convenient macro for linking to all libraries:

project(MyHello)
cmake_minimum_required(VERSION 2.8.12)
include(${CMAKE_BINARY_DIR}/conanbuildinfo_multi.cmake)
conan_basic_setup()
add_executable(say_hello main.cpp)
conan_target_link_libraries(say_hello)

With this approach, the end user can open the generated IDE project and switch among both configurations, building the project, or from the command line:

$ cmake --build . --config Release
# And without having to conan install again, or do anything else
$ cmake --build . --config Debug

Creating packages

The cmake_multi generator is just for consumption. It cannot be used to create packages. If you want to be able to both use the cmake_multi generator to install dependencies and build your project but also to create packages from that code, you need to specify the regular cmake generator for package creation, and prepare the CMakeLists.txt accordingly, something like:

project(MyHello)
cmake_minimum_required(VERSION 2.8.12)
if(EXISTS ${CMAKE_BINARY_DIR}/conanbuildinfo_multi.cmake)
  include(${CMAKE_BINARY_DIR}/conanbuildinfo_multi.cmake)
else()
  include(${CMAKE_BINARY_DIR}/conanbuildinfo.cmake)
endif()
conan_basic_setup()
add_executable(say_hello main.cpp)
conan_target_link_libraries(say_hello)

Then, make sure that the generator cmake_multi is not specified in the conanfiles, but the users specify it in the command line while installing dependencies:

$ conan install . -g cmake_multi

See also:

Check the section Reference/Generators/cmake to read more about this generator.
**cmake_paths generator**

This generator is especially useful if you are using CMake based only on the `find_package` feature to locate the dependencies.

The `cmake_paths` generator creates a file named `conan_paths.cmake` declaring:

- `CMAKE_MODULE_PATH` with the folders of the required packages, to allow CMake to locate the included cmake scripts and `FindXXX.cmake` files. The folder containing the `conan_paths.cmake` (self.install_folder when used in a recipe) is also included, so any custom file will be located too. Check `cmake_find_package_generator` generator.

- `CMAKE_PREFIX_PATH` used by `find_library()` to locate library files (.a, .lib, .so, .dll) in your packages and `find_dependency()` to locate the transitive dependencies.

```plaintext
Listing 2: conanfile.txt

[requires]
zlib/1.2.11@conan/stable...

[generators]
cmake_paths
```

```plaintext
Listing 3: CMakeList.txt

cmake_minimum_required(VERSION 3.0)
project(helloworld)
add_executable(helloworld hello.c)
find_package(Zlib)
if(ZLIB_FOUND)
  include_directories(${ZLIB_INCLUDE_DIRS})
  target_link_libraries (helloworld ${ZLIB_LIBRARIES})
endif()
```

In the example above, the `zlib/1.2.11@conan/stable` package is not packaging a custom `FindZLIB.cmake` file, but the `FindZLIB.cmake` included in the CMake installation directory (/Modules) will locate the zlib library from the Conan package because of the `CMAKE_PREFIX_PATH` used by the `find_library()`.

If the `zlib/1.2.11@conan/stable` would have included a custom `FindZLIB.cmake` in the package root folder or any declared `self.cpp_info.builddirs`, it would have been located because of the `CMAKE_MODULE_PATH` variable.

**Included as a toolchain**

You can use the `conan_paths.cmake` as a toolchain without modifying your `CMakeLists.txt` file:

```plaintext
$ mkdir build && cd build
$ conan install ..
$ cmake .. -DCMAKE_TOOLCHAIN_FILE=conan_paths.cmake -G "Unix Makefiles" -DCMAKE_BUILD_TYPE=Release
$ cmake --build .
```

---

**12.1. Build systems**  

147
Included using the \texttt{CMAKE\_PROJECT\_<PROJECT\-NAME>\_INCLUDE}

With \texttt{CMAKE\_PROJECT\_<PROJECT\-NAME>\_INCLUDE} you can specify a file to be included by the \texttt{project()} command. If you already have a toolchain file you can use this variable to include the \texttt{conan\_paths.cmake} and insert your toolchain with the \texttt{CMAKE\_TOOLCHAIN\_FILE}.

```bash
$ mkdir build && cd build
$ conan install ..
$ cmake .. -G "Unix Makefiles" -DCMAKE\_BUILD\_TYPE=Release -DCMAKE\_PROJECT\_helloworld\_\_\_\_INCLUDE=build/conan\_paths.cmake
$ cmake --build .
```

Included in your \texttt{CMakeLists.txt}

Listing 4: \texttt{CMakeList.txt}

```cmake
cmake\_minimum\_required(VERSION 3.0)
project(helloworld)

include($\{CMAKE\_BINARY\_DIR\}/conan\_paths.cmake)

add\_executable(helloworld hello.c)

find\_package(zlib)

if(ZLIB\_FOUND)
    include\_directories($\{ZLIB\_INCLUDE\_DIRS\})
    target\_link\_libraries (helloworld $\{ZLIB\_LIBRARIES\})
endif()
```

```bash
$ mkdir build && cd build
$ conan install ..
$ cmake .. -G "Unix Makefiles" -DCMAKE\_BUILD\_TYPE=Release
$ cmake --build .
```

See also:

Check the section \texttt{cmake\_paths} to read more about this generator.

\textbf{Note:} The \texttt{CMAKE\_MODULE\_PATH} and \texttt{CMAKE\_PREFIX\_PATH} contain the paths to the \texttt{build\_dirs} of every required package. By default the root package folder is the only declared \texttt{build\_dirs} directory. Check \texttt{cpp\_info}.

\texttt{cmake\_find\_package} generator

This generator is especially useful if you are using CMake using the \texttt{find\_package} feature to locate the dependencies.

The \texttt{cmake\_find\_package} generator creates a file for each requirement specified in a conanfile.

The name of the files follows the pattern \texttt{Find<package\_name>\_\_\_.cmake}. So for the zlib/1.2.11@conan/stable package, a \texttt{Findzlib\_\_\_.cmake} file will be generated.
In a conanfile.py

Listing 5: conanfile.py

```python
from conans import ConanFile, tools

class LibConan(ConanFile):
    requires = "zlib/1.2.11@conan/stable"
    generators = "cmake_find_package"

    def build(self):
        cmake = CMake(self)
        # it will find the packages by using our auto-generated FindXXX.cmake files
        cmake.configure()
        cmake.build()
```

In the previous example, the CMake build helper will automatically adjust the CMAKE_MODULE_PATH to the conanfile.install_folder, where the generated Find<package_name>.cmake is.

In the CMakeList.txt you do not need to specify or include anything related with Conan at all; just rely on the find_package feature:

Listing 6: CMakeList.txt

```cmake
cmake_minimum_required(VERSION 3.0)
project(helloworld)
add_executable(helloworld hello.c)
find_package(zlib)

# Global approach
if(zlib_FOUND)
    include_directories(${zlib_INCLUDE_DIRS})
    target_link_libraries (helloworld ${zlib_LIBRARIES})
endif()

# Modern CMake targets approach
if(TARGET zlib::zlib)
    target_link_libraries(helloworld zlib::zlib)
endif()
```

$ conan create . user/channel

lib/1.0@user/channel: Calling build()
-- The C compiler identification is AppleClang 9.1.0.9020039
...
-- Conan: Using autogenerate Findzlib.cmake
-- Found: /Users/user/.conan/data/zlib/1.2.11/conan/stable/package/
  0eaf3bfbcb94fb6d2c8f230d052d75c6c1a57a4ce/lib/libz.a
lib/1.0@user/channel: Package '72bce3af445a371b892525bc8701d96c568ead8b' created

In a conanfile.txt

If you are using a conanfile.txt file in your project, instead of a conanfile.py, this generator can be used together with the cmake_paths generator to adjust the CMAKE_MODULE_PATH and CMAKE_PREFIX_PATH variables.

12.1. Build systems
automatically and let CMake locate the generated `Find<package_name>.cmake` files.

With `cmake_paths`:

Listing 7: conanfile.txt

```conan
[requires]
zlib/1.2.11@conan/stable
...
[generators]
cmake_find_package
cmake_paths
```

Listing 8: CMakeList.txt

```cmake
cmake_minimum_required(VERSION 3.0)
project(helloworld)
include(${CMAKE_BINARY_DIR}/conan_paths.cmake)
add_executable(helloworld hello.c)
find_package(zlib)

# Global approach
if(zlib_FOUND)
    include_directories(${zlib_INCLUDE_DIRS})
    target_link_libraries (helloworld ${zlib_LIBRARIES})
endif()

# Modern CMake targets approach
if(TARGET zlib::zlib)
    target_link_libraries(helloworld zlib::zlib)
endif()
```

$ mkdir build && cd build
$ conan install ..
$ cmake .. -G "Unix Makefiles" -DCMAKE_BUILD_TYPE=Release
  -- Conan: Using autogenerated Findzlib.cmake
  -- Found: /Users/user/.conan/data/zlib/1.2.11/conan/stable/package/
  → 0eaf3bfbc94fb6d2c8f230d052d75c6c1a57a4ce/lib/libz.a
  ...
$ cmake --build .

Or you can also adjust `CMAKE_MODULE_PATH` and `CMAKE_PREFIX_PATH` manually.

Without `cmake_paths`, adjusting the variables manually:

Listing 9: conanfile.txt

```conan
[requires]
zlib/1.2.11@conan/stable
...
[generators]
cmake_find_package
```
Listing 10: CMakeList.txt

```cmake
cmake_minimum_required(VERSION 3.0)
project(helloworld)
set(CMAKE_MODULE_PATH ${CMAKE_BINARY_DIR} ${CMAKE_MODULE_PATH})
set(CMAKE_PREFIX_PATH ${CMAKE_BINARY_DIR} ${CMAKE_PREFIX_PATH})
add_executable(helloworld hello.c)
find_package(zlib)

# Global approach
if(zlib_FOUND)
    include_directories(${zlib_INCLUDE_DIRS})
    target_link_libraries (helloworld ${zlib_LIBRARIES})
endif()

# Modern CMake targets approach
if(TARGET zlib::zlib)
    target_link_libraries(helloworld zlib::zlib)
endif()
```

See also:
Check the section `cmake_find_package` to read more about this generator and the adjusted CMake variables/targets.

**cmake_find_package_multi**

**Warning:** This is an experimental feature subject to breaking changes in future releases.

This generator is similar to the `cmake_find_package` generator but it allows working with multi-configuration projects like Visual Studio with both Debug and Release. But there are some differences:

- Only works with CMake > 3.0
- It doesn’t generate `FindXXX.cmake` modules but `XXXConfig.cmake` files.
- The “global” approach is not supported, only “modern” CMake by using targets.

**Usage**

```bash
$ conan install . -g cmake_find_package_multi -s build_type=Debug
$ conan install . -g cmake_find_package_multi -s build_type=Release
```

These commands will generate several files for each dependency in your graph, including a `XXXConfig.cmake` that can be located by the CMake `find_package(XXX)` command, with XXX as the package name.

The name of the files follows the pattern `<package_name>Config.cmake`. So for the zlib/1.2.11@conan/stable package, a zlibConfig.cmake file will be generated.

See also:
Check the section `cmake_find_package_multi` to read more about this generator and the adjusted CMake variables/targets.
**Build automation**

You can invoke CMake from your conanfile.py file and automate the build of your library/project. Conan provides a CMake() helper. This helper is useful in calling the `cmake` command both for creating Conan packages or automating your project build with the `conan build` command. The CMake() helper will take into account your settings in order to automatically set definitions and a generator according to your compiler, build_type, etc.

See also:
Check the section *Building with CMake*.

**Find Packages**

If a FindXXX.cmake file for the library you are packaging is already available, it should work automatically.

Variables `CMAKE_INCLUDE_PATH` and `CMAKE_LIBRARY_PATH` are set with the requirements paths. The CMake `find_library` function will be able to locate the libraries in the package’s folders.

So, you can use `find_package` normally:

```
project(MyHello)
cmake_minimum_required(VERSION 2.8.12)

include(conanbuildinfo.cmake)
conan_basic_setup()

find_package("ZLIB")

if(ZLIB_FOUND)
    add_executable(enough enough.c)
    include_directories(${ZLIB_INCLUDE_DIRS})
    target_link_libraries(enough ${ZLIB_LIBRARIES})
else()
    message(FATAL_ERROR "Zlib not found")
endif()
```

In addition to automatic `find_package` support, `CMAKE_MODULE_PATH` variable is set with the requirements root package paths. You can override the default behavior of any find_package() by creating a `findXXX.cmake` file in your package.

**Creating a custom FindXXX.cmake file**

Sometimes the “official” CMake FindXXX.cmake scripts are not ready to find our libraries (unsupported library names for specific settings, fixed installation directories like `C:\OpenSSL`, etc.) Or maybe there is no “official” CMake script for our library.

In these cases we can provide a custom `FindXXX.cmake` file in our Conan packages.

1. Create a file named FindXXX.cmake and save it in your Conan package root folder, where XXX is the name of the library that we will use in the `find_package` CMake function. For example, we create a `FindZLIB.cmake` and use `find_package(ZLIB)`. We recommend copying the original FindXXX.cmake file from Kitware (folder Modules/FindXXX.cmake), if available, and modifying it to help find our library files, but it depends a lot; maybe you are interested in creating a new one.

If it’s not provided, you can create a basic one. Take a look at this example with the ZLIB library:

`FindZLIB.cmake`
In the first line we find the path where the headers should be found. We suggest the CONAN_INCLUDE_DIRS_XXX. Then repeat for the library names with CONAN_LIBS_XXX and the paths where the libs are CONAN_LIB_DIRS_XXX.

2. In your conanfile.py file add the FindXXX.cmake to the exports_sources field:

```python
class HelloConan(ConanFile):
    name = "Hello"
    version = "0.1"
    ...
    exports_sources = ["FindXXX.cmake"]
```

3. In the package method, copy the FindXXX.cmake file to the root:

```python
class HelloConan(ConanFile):
    name = "Hello"
    version = "0.1"
    ...
    exports_sources = ["FindXXX.cmake"]

    def package(self):
        ...
        self.copy("FindXXX.cmake", ".", ".")
```

### 12.1.2 MSBuild (Visual Studio)

Conan can be integrated with MSBuild, the build system of Visual Studio in two different ways:

- Using the cmake generator to create a conanbuildinfo.cmake file.
- Using the visual_studio generator to create a conanbuildinfo.props file.

**With CMake**

Use the cmake generator or cmake_multi if you are using CMake to machine-generate your Visual Studio projects. Check the Generators section to read about the cmake generator. Check the official CMake docs to find out more about generating Visual Studio projects with CMake.

However, beware of some current CMake limitations, such as not dealing well with find-packages, because CMake doesn’t know how to handle finding both debug and release packages.
Note: If you want to use the Visual Studio 2017 + CMake integration, check this how-to

With `visual_studio` generator

Use the `visual_studio` generator, or `visual_studio_multi`, if you are maintaining your Visual Studio projects, and want to use Conan to tell Visual Studio how to find your third-party dependencies.

You can use the `visual_studio` generator to manage your requirements via your Visual Studio project.

This generator creates a Visual Studio project properties file, with all the include paths, lib paths, libs, flags etc., that can be imported in your project.

Open `conanfile.txt` and change (or add) the `visual_studio` generator:

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

[generators]
visual_studio
```

Install the requirements:

```
$ conan install .
```

Go to your Visual Studio project, and open the Property Manager (usually in View -> Other Windows -> Property Manager).
Click the + icon and select the generated `conanbuildinfo.props` file:

Build your project as usual.

**Note:** Remember to set your project’s architecture and build type accordingly, explicitly or implicitly, when issuing the `conan install` command. If these values don’t match, your build will probably fail.

**e.g.** `Release/x64`

See also:

Check `visual_studio` for the complete reference.
Calling Visual Studio compiler

You can call the Visual Studio compiler from your build() method using the VisualStudioBuildEnvironment and the tools.vcvars_command().

Check the MSBuild section for more info.

Build an existing Visual Studio project

You can build an existing Visual Studio from your build() method using the MSBuild() build helper.

```python
from conans import ConanFile, MSBuild

class ExampleConan(ConanFile):
    ...
    def build(self):
        msbuild = MSBuild(self)
        msbuild.build("MyProject.sln")
```

Toolsets

You can use the sub-setting toolset of the Visual Studio compiler to specify a custom toolset. It will be automatically applied when using the CMake() and MSBuild() build helpers. The toolset can also be specified manually in these build helpers with the toolset parameter.

By default, Conan will not generate a new binary package if the specified compiler.toolset matches an already generated package for the corresponding compiler.version. Check the package_id() reference to learn more.

See also:

Check the CMake() reference section for more info.

12.1.3 Autotools: configure/make

If you are using configure/make you can use the AutoToolsBuildEnvironment helper. This helper sets LIBS, LDFLAGS, CFLAGS, CXXFLAGS and CPPFLAGS environment variables based on your requirements.

Check Building with Autotools for more info.

12.1.4 Ninja, NMake, Borland

These build systems still don’t have a Conan generator for using them natively. However, if you are using cmake, you can instruct Conan to use them instead of the default generator (typically Unix Makefiles) defining the environment variable CONAN_CMAKE_GENERATOR.

Read more about this variable in Environment variables.

12.1.5 pkg-config and .pc files

If you are creating a Conan package for a library (A) and the build system uses .pc files to locate its dependencies (B and C) that are Conan packages too, you can follow different approaches.
The main issue to address is the absolute paths. When a user installs a package in the local cache, the directory will probably be different from the directory where the package was created. This could be because of the different computer, the change in Conan home directory or even a different user or channel:

For example, in the machine where the packages were created:

```
/home/user/lasote/.data/storage/zlib/1.2.11/conan/stable
```

In the machine where the library is being reused:

```
/custom/dir/.data/storage/zlib/1.2.11/conan/testing
```

You can see that `.pc` files containing absolute paths won’t work with locating the dependencies.

Example of a `.pc` file with an absolute path:

```
prefix=/Users/lasote/.conan/data/zlib/1.2.11/lasote/stable/package/
˓
b5d68b3533204ad67e01fa587ad28fb8ce010527
exec_prefix=${prefix}
libdir=${exec_prefix}/lib
sharedlibdir=${libdir}
includedir=${prefix}/include

Name: zlib
Description: zlib compression library
Version: 1.2.11

Requires:
Libs: -L${libdir} -L${sharedlibdir} -lz
Cflags: -I${includedir}
```

To solve this problem there are different approaches that can be followed.

**Approach 1: Import and patch the prefix in the `.pc` files**

In this approach your library A will import to a local directory the `.pc` files from B and C, then, as they will contain absolute paths, the recipe for A will patch the paths to match the current installation directory.

You will need to package the `.pc` files from your dependencies. You can adjust the `PKG_CONFIG_PATH` to let `pkg-config` tool locate them.

```python
import os
from conans import ConanFile, tools

class LibAConan(ConanFile):
    name = "libA"
    version = "1.0"
    settings = "os", "compiler", "build_type", "arch"
    exports_sources = "*.cpp"
    requires = "libB/1.0@conan/stable"

    def build(self):
        lib_b_path = self.deps_cpp_info["libB"].rootpath
        copyfile(os.path.join(lib_b_path, "libB.pc"), "libB.pc")
        # Patch copied file with the libB path
        tools.replace_prefix_in_pc_file("libB.pc", lib_b_path)
```

(continues on next page)
with tools.environment_append({"PKG_CONFIG_PATH": os.getcwd()})
    # CALL YOUR BUILD SYSTEM (configure, make etc)
    # E.g., self.run('g++ main.cpp $(pkg-config libB --libs --cflags) -o main')

Approach 2: Prepare and package .pc files before packaging them

With this approach you will patch the .pc files from B and C before packaging them. The goal is to replace the absolute path (the variable part of the path) with a variable placeholder. Then in the consumer package A, declare the variable using --define-variable when calling the pkg-config command.

This approach is cleaner than approach 1, because the packaged files are already prepared to be reused with or without Conan by declaring the needed variable. And it’s unneeded to import the .pc files to the consumer package. However, you need B and C libraries to package the .pc files correctly.

Library B recipe (preparing the .pc file):

```python
from conans import ConanFile, tools

class LibBConan(ConanFile):
    ....
    def build(self):
        ...
        tools.replace_prefix_in_pc_file("mypcfile.pc", "${package_root_path_lib_b}")
        
    def package(self):
        self.copy(pattern="*.pc", dst='', keep_path=False)
```

Library A recipe (importing and consuming .pc file):

```python
class LibAConan(ConanFile):
    ....
    
    requires = "libB/1.0@conan/stable, libC/1.0@conan/stable"
    
    def build(self):
        
        args = '--define-variable package_root_path_lib_b=%s' % self.deps_cpp_info['libB'].rootpath
        args += ' --define-variable package_root_path_lib_c=%s' % self.deps_cpp_info['libC'].rootpath
        pkgconfig_exec = 'pkg-config ' + args
        
        vars = {'PKG_CONFIG': pkgconfig_exec, 'PKG_CONFIG_PATH': self.deps_cpp_info['libB'].rootpath, self.deps_cpp_info['libC'].rootpath}  
        
        with tools.environment_append(vars):
            # Call autotools (.configure ./make, will read PKG_CONFIG)
            # Or directly declare the variables:
            # self.run('g++ main.cpp $(pkg-config $s libB --libs --cflags) -o main' % vars['PKG_CONFIG_PATH'], args)
```
**Approach 3: Use `--define-prefix`**

If you have available `pkg-config >= 0.29` and you have only one dependency, you can directly use the `--define-prefix` option to declare a custom prefix variable. With this approach you won’t need to patch anything, just declare the correct variable.

**Approach 4: Use `PKG_CONFIG_$PACKAGE_$VARIABLE`**

If you have `pkg-config >= 0.29.1` available, you can manage multiple dependencies declaring variables with the prefixes:

```python
class LibAConan(ConanFile):
    ....
    requires = "libB/1.0@conan/stable, libC/1.0@conan/stable"

    def build(self):
        vars = {'PKG_CONFIG_libB_PREFIX': self.deps_cpp_info["libB"].rootpath,
                'PKG_CONFIG_libC_PREFIX': self.deps_cpp_info["libC"].rootpath,
                'PKG_CONFIG_PATH': "$s:$s" % (self.deps_cpp_info["libB"].rootpath,
                                              self.deps_cpp_info["libC"].rootpath)}

        with tools.environment_append(vars):
            # Call the build system
```

**Approach 5: Use the `pkg_config` generator**

If you use `package_info()` in library B and library C, and specify all the library names and any other needed flag, you can use the `pkg_config` generator for library A. Those files don’t need to be patched, because they are dynamically generated with the correct path.

So it can be a good solution in case you are building library A with a build system that manages `.pc` files like Meson Build or AutoTools:

**Meson Build**

```python
from conans import ConanFile, tools, Meson
import os

class ConanFileToolsTest(ConanFile):
    generators = "pkg_config"
    requires = "LIB_A/0.1@conan/stable"
    settings = "os", "compiler", "build_type"

    def build(self):
        meson = Meson(self)
        meson.configure()
        meson.build()
```

**Autotools**

```python
from conans import ConanFile, tools, AutoToolsBuildEnvironment
import os
```

(continues on next page)
class ConanFileToolsTest(ConanFile):
    generators = "pkg_config"
    requires = "LIB_A/0.1@conan/stable"
    settings = "os", "compiler", "build_type"

    def build(self):
        autotools = AutoToolsBuildEnvironment(self)
        # When using the pkg_config generator, self.build_folder will be added to PKG_CONFIG_PATH
        # so pkg_config will be able to locate the generated pc files from the requires (LIB_A)
        autotools.configure()
        autotools.make()

See also:
Check the tools.PkgConfig(), a wrapper of the pkg-config tool that allows to extract flags, library paths, etc. for any .pc file.

Caution:  This generator is deprecated in favor of the b2 generator. See generator b2.

With this generator boost-build you can generate a project-root.jam file to be used with your Boost Build system.
Check the generator boost-build

With this generator b2 you can generate a conanbuildinfo.jam file to be used with your B2 system.
Check the generator b2

12.1.8 QMake

The qmake generator will generate a conanbuildinfo.pri file that can be used for your qmake builds.

$ conan install . -g qmake

Add conan_basic_setup to CONFIG and include the file in your existing project .pro file:
Listing 11: `yourproject.pro`

    ...  
    CONFIG += conan_basic_setup 
    include(conanbuildinfo.pri)  

This will include all the statements in `conanbuildinfo.pri` in your project. Include paths, libraries, defines, etc. will be set up for all requirements you have defined as dependencies in a `conanfile.txt`.

If you’d prefer to manually add the variables for each dependency, you can do so by skipping the `CONFIG` statement and only including `conanbuildinfo.pri`:

Listing 12: `yourproject.pro`

    # ...  
    include(conanbuildinfo.pri)  
    # you may now modify your variables manually for each library, such as  
    # INCLUDEPATH += CONAN_INCLUDEPATH_POCO

The qmake generator allows multi-configuration packages, i.e. packages that contains both Debug and Release artifacts.

**Example**

**Tip:** This complete example is stored in [https://github.com/memsharded/qmake_example](https://github.com/memsharded/qmake_example)

This example project will depend on a multi-configuration (Debug/Release) “Hello World” package. It should be installed first:

```bash
$ git clone https://github.com/memsharded/hello_multi_config  
$ cd hello_multi_config  
$ conan create . memsharded/testing  
Hello/0.1@memsharded/testing export: Copied 1 '.txt' file: CMakeLists.txt  
Hello/0.1@memsharded/testing export: Copied 1 '.cpp' file: hello.cpp  
Hello/0.1@memsharded/testing export: Copied 1 '.h' file: hello.h  
Hello/0.1@memsharded/testing: A new conanfile.py version was exported
```

This hello package is created with CMake, but that doesn’t matter for this example, as it can be consumed from a qmake project with the configuration showed before.

Now let’s get the qmake project and install its `Hello/0.1@memsharded/testing` dependency:

```bash
$ git clone https://github.com/memsharded/qmake_example  
$ cd qmake_example  
$ conan install .  
PROJECT: Installing C:\Users\memsharded\qmake_example\conanfile.txt  
Requirements  
  Hello/0.1@memsharded/testing from local cache - Cache  
Packages  
  Hello/0.1@memsharded/testing:15af85373a5688417675aa1e5065700263bf257e - Cache
```

(continues on next page)
Hello/0.1@memsharded/testing: Already installed!
PROJECT: Generator qmake created conanbuildinfo.pri
PROJECT: Generator txt created conanbuildinfo.txt
PROJECT: Generated conaninfo.txt

As you can see, we got the dependency information in the conanbuildinfo.pri file. You can inspect the file to see the variables generated. Now let’s build the project for Release and then for Debug:

```bash
$ qmake
$ make
$ ./helloworld
> Hello World Release!

# now let’s build the Debug one
$ make clean
$ qmake CONFIG+=debug
$ make
$ ./helloworld
> Hello World Debug!
```

See also:

Check the complete reference of the qmake generator.

---

12.1.9 Premake

Since Conan 1.9.0 the premake generator is built-in and works with premake5, so the following should be enough to use it:

```make
[generators]
p remake
```

Example

We are going to use the same example from Getting Started, a MD5 Encrypter app.

This is the main source file for it:

Listing 13: main.cpp

```cpp
#include "Poco/MD5Engine.h"
#include "Poco/DigestStream.h"

#include <iostream>

int main(int argc, char** argv)
{
    Poco::MD5Engine md5;
    Poco::DigestOutputStream ds(md5);

    (continues on next page)```
As this project relies on the Poco Libraries, we are going to create a `conanfile.txt` with our requirement and also declare the Premake generator:

**Listing 14: conanfile.txt**

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

[generators]
premake
```

In order to use the new generator within your project, use the following Premake script as a reference:

**Listing 15: premake5.lua**

```lua
-- premake5.lua
include("conanbuildinfo.premake.lua")

workspace("ConanPremakeDemo")
conan_basic_setup()

project "ConanPremakeDemo"
kind "ConsoleApp"
language "C++"
targetdir "bin/%{cfg.buildcfg}"

linkoptions { conan_exelinkflags }
files ( "**.h", "**.cpp" )

filter "configurations:Debug"
defines ( "DEBUG" )
symbols "On"

filter "configurations:Release"
defines ( "NDEBUG" )
optimize "On"
```

Now we are going to let Conan retrieve the dependencies and generate the dependency information in a `conanbuildinfo.lua`:

```bash
$ conan install .
```

Then let's call premake to generate our project:

- Use this command for Windows Visual Studio:
  ```bash
  $ premake5 vs2017 # Generates a .sln
  ```
- Use this command for Linux or macOS:
Now you can build your project with Visual Studio or Make.

**See also:**
Check the complete reference of the *premake* generator.

### 12.1.10 Make

Conan provides integration with plain Makefiles by means of the *make* generator. If you are using *Makefile* to build your project you could get the information of the dependencies in a *conanbuildinfo.mak* file. All you have to do is indicate the generator like this:

Listing 16: *conanfile.txt*

```text
[generators]
make
```

Listing 17: *conanfile.py*

```python
class MyConan(ConanFile):
...  
generators = "make"
```

**Example**

We are going to use the same example from *Getting Started*, a MD5 Encrypter app.

This is the main source file for it:

Listing 18: *main.cpp*

```cpp
#include "Poco/MD5Engine.h"
#include "Poco/DigestStream.h"
#include <iostream>

int main(int argc, char** argv)
{
    Poco::MD5Engine md5;
    Poco::DigestOutputStream ds(md5);
    ds << "abcdefghijklmnopqrstuvwxyz";
    ds.close();
    std::cout << Poco::DigestEngine::digestToHex(md5.digest()) << std::endl;
    return 0;
}
```

As this project relies on the Poco Libraries we are going to create a *conanfile.txt* with our requirement and also declare the Make generator:
In order to use this generator within your project, use the following Makefile as a reference:

```makefile
#include conanbuildinfo.mak

#----------------------------------------
# Make variables for a sample App
#----------------------------------------

CXX_SRCS = \ main.cpp
CXX_OBJ_FILES = \ main.o
EXE_FILENAME = \ main

#----------------------------------------
# Prepare flags from variables
#----------------------------------------

CFLAGS += $(CONAN_CFLAGS)
CXXFLAGS += $(CONAN_CXXFLAGS)
CPPFLAGS += $(addprefix -I, $(CONAN_INCLUDE_DIRS))
CPPFLAGS += $(addprefix -D, $(CONAN_DEFINES))
LDFLAGS += $(addprefix -L, $(CONAN_LIB_DIRS))
LDLIBS += $(addprefix -l, $(CONAN_LIBS))

#----------------------------------------
# Make Commands
#----------------------------------------

COMPILE_CXX_COMMAND ?= \\ g++ -c $(CPPFLAGS) $(CXXFLAGS) $< -o $@

CREATE_EXE_COMMAND ?= \\ g++ $(CXX_OBJ_FILES) \ $(CXXFLAGS) $(LDFLAGS) $(LDLIBS) \ -o $(EXE_FILENAME)

#----------------------------------------
# Make Rules
#----------------------------------------
```

(continues on next page)
Now we are going to let Conan retrieve the dependencies and generate the dependency information in a conanbuild-info.mak:

```
$ conan install .
```

Then let’s call `make` to generate our project:

```
$ make exe
```

Now you can run your application with `./main`.

**See also:**
Check the complete reference of the *Make generator*.

### 12.1.11 qbs

Conan provides a *qbs* generator, which will generate a `conanbuildinfo.qbs` file that can be used for your qbs builds.

Add `conanbuildinfo.qbs` as a reference on the project level and a `Depends` item with the name `conanbuildinfo`:

```qbs
yourproject.qbs
import qbs
Project {
    references: ["conanbuildinfo.qbs"]
    Product {
        type: "application"
        consoleApplication: true
        files: [
            "conanfile.txt",
            "main.cpp",
        ]
        Depends { name: "cpp" }  
        Depends { name: "ConanBasicSetup" }
    }
}
```

This will include the product called *ConanBasicSetup* which holds all the necessary settings for all your dependencies.

If you’d prefer to manually add each dependency, just replace `ConanBasicSetup` with the dependency you would like to include. You may also specify multiple dependencies:

```qbs
yourproject.qbs
```
import qbs

Project {
  references: ["conanbuildinfo.qbs"]
  Product {
    type: "application"
    consoleApplication: true
    files: [
      "conanfile.txt",
      "main.cpp",
    ]
    Depends { name: "cpp" }
    Depends { name: "catch" }
    Depends { name: "Poco" }
  }
}

See also:

Check the Reference/Generators/qbs section for get more details.

12.1.12 Meson Build

If you are using Meson Build as your library build system, you can use the Meson build helper. This helper has .configure() and .build() methods available to ease the call to Meson build system. It also will automatically take the pc files of your dependencies when using the pkg_config generator.

Check Building with Meson Build for more info.

12.1.13 SCons

SCons can be used both to generate and consume Conan packages via the scons generator. The package recipe build() method could be similar to:

class PkgConan(ConanFile):
  settings = 'os', 'compiler', 'build_type', 'arch'
  requires = ['Hello/1.0@user/stable']
  generators = "scons"
  ...

  def build(self):
    debug_opt = '--debug-build' if self.settings.build_type == 'Debug' else ''
    os.makedirs("build")
    # FIXME: Compiler, version, arch are hardcoded, not parametrized
    with tools.chdir("build"):
      self.run('scons -C {}\src {}'.format(self.source_folder, debug_opt))

(continues on next page)
The `SConscript` build script can load the generated `SConscript_conan` file that contains the information of the dependencies, and use it to build

```python
conan = SConscript('{}\SConscript_conan'.format(build_path_relative_to_sconstruct))
if not conan:
    print("File `SConscript_conan` is missing.")
    print("It should be generated by running `conan install`.")
    sys.exit(1)

flags = conan["conan"]
version = flags.pop("VERSION")
env.MergeFlags(flags)
env.Library("hello", "hello.cpp")
```

A complete example with a `test_package` that uses SCons too is available in the following GitHub repository. Give it a try!

```
$ git clone https://github.com/memsharded/conan-scons-template
$ cd conan-scons-template
$ conan create . demo/testing
>Hello World Release!
$ conan create . demo/testing -s build_type=Debug
>Hello World Debug!
```

### 12.1.14 Compilers on command line

The `compiler_args` generator creates a file named `conanbuildinfo.args` containing command line arguments to invoke `gcc`, `clang` or `cl` (Visual Studio) compiler.

Now we are going to compile the `getting started` example using `compiler_args` instead of the `cmake` generator.

Open `conanfile.txt` and change (or add) `compiler_args` generator:

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

[generators]
compiler_args
```

Install the requirements (from the mytimer/build folder):

```
$ conan install ..
```

**Note:** Remember, if you don’t specify settings in the `install command` with `-s`, Conan will use the detected defaults. You can always change them by editing the `~/.conan/profiles/default` or override them with “-s” parameters.

The generated `conanbuildinfo.args` show:
This is hard to read, but those are just the **compiler_args** parameters needed to compile our program:

- **-I** options with headers directories
- **-L** for libraries directories
- **-l** for library names
- and so on... see the complete reference here

It's almost the same information we can see in `conanbuildinfo.cmake` and many other generators' files.

Run:

```
$ mkdir bin
$ g++ ../timer.cpp @conanbuildinfo.args -std=c++14 -o bin/timer
```

**Note:** “@conanbuildinfo.args” appends all the file contents to g++ command parameters

```
$ ./bin/timer
Callback called after 250 milliseconds.
...
```

To invoke cl (Visual Studio compiler):

```
$ cl /EHsc timer.cpp @conanbuildinfo.args
```

You can also use the generator within your `build()` method of your conanfile.py.

Check the Reference, generators, compiler_args section for more info.
12.2 IDEs

You can develop both the recipes and your libraries using your IDE.

12.2.1 Visual Studio

Conan Extension for Visual Studio

Thanks to the invaluable help of our community we manage to develop and maintain a free extension for Visual Studio in the Microsoft Marketplace, it is called Conan Extension for Visual Studio and it provides integration with Conan using the Visual Studio generators.

You can install it into your IDE using the Extensions manager and start using it right away. This extension will look for a `conanfile.py` (or `conanfile.txt`) and retrieve the requirements declared in it that match your build configuration (it will build them from sources if no binaries are available).

**Note: Location of the conanfile**

In version 1.0 of the extension, the algorithm to look for the `conanfile.py` (preferred) or `conanfile.txt` is very naïve: It will start looking for those files in the directory where the Visual Studio project file is located and then it will walk recursively into parent directories to look for them.

The extension creates a property sheet file and adds it to the project, so all the information from the dependencies handled by Conan should be added (as inherited properties) to those already available in your projects.

At this moment (release v1.0.x) the extension is under heavy development, some behaviors may change and new features will be added. You can subscribe to its repository to stay updated and, of course, any feedback about it will be more than welcome.
General Integration

Check the `MSBuild()` integration, that contains information about Build Helpers and generators to be used with Visual Studio.

12.2.2 CLion

Conan-Clion plugin

There is an official Jetbrains plugin Conan plugin for Clion.

You can read how to use it in the following blog post

General Integration

CLion uses CMake as the build system of projects, so you can use the CMake generator to manage your requirements in your CLion project.

Just include the `conanbuildinfo.cmake` this way:
If the `conanbuildinfo.cmake` file is not found, it will print a warning message in the `Messages` console of your CLion IDE.

**Using packages in a CLion project**

Let see an example of how to consume Conan packages in a CLion project. We are going to require and use the `zlib` Conan package.

1. Create a new CLion project

2. Edit the `CMakeLists.txt` file and add the following lines:

```cmake
if(EXISTS ${CMAKE_BINARY_DIR}/conanbuildinfo.cmake)
    include(${CMAKE_BINARY_DIR}/conanbuildinfo.cmake)
    conan_basic_setup()
else()
    message(WARNING "The file conanbuildinfo.cmake doesn't exist, you have to run conan install first")
endif()
```

This will allow you to use the `zlib` package in your project.
3. CLion will reload your CMake project and you will be able to see a Warning in the console, because the conanbuildinfo.cmake file still doesn’t exist:

4. Create a conanfile.txt with all your requirements and use the cmake generator. In this case we only require the zlib library from a Conan package:

   ```text
   [requires]
   zlib/1.2.11@conan/stable
   
   [generators]
   cmake
   ```

5. Now you can run `conan install` for debug in the cmake-build-debug folder to install your requirements and generate the conanbuildinfo.cmake file there:

   ```bash
   $ conan install . -s build_type=Debug --install-folder=cmake-build-debug
   ```

6. Repeat the last step if you have the release build types configured in your CLion IDE, but change the build_type setting accordingly:

   ```bash
   $ conan install . -s build_type=Release --install-folder=cmake-build-release
   ```

7. Now reconfigure your CLion project. The Warning message is not shown anymore:
8. Open the `library.cpp` file and include `zlib.h`. If you follow the link, you can see that CLion automatically detects the `zlib.h` header file from the local Conan cache.

9. Build your project normally using your CLion IDE:

You can check a complete example of a CLion project reusing Conan packages in this github repository: lasote/clion-conan-consumer.

**Creating Conan packages in a CLion project**

Now we are going to see how to create a Conan package from the previous library.

1. Create a new CLion project
2. Edit the CMakeLists.txt file and add the following lines:

```cmake
if(EXISTS ${CMAKE_BINARY_DIR}/conanbuildinfo.cmake)
    include(${CMAKE_BINARY_DIR}/conanbuildinfo.cmake)
    conan_basic_setup()
else()
    message(WARNING "The file conanbuildinfo.cmake doesn't exist, you have to run conan install first")
endif()
```

3. Create a conanfile.py file. It’s recommended to use the `conan new` command.

```
$ conan new mylibrary/1.0@myuser/channel
```

Edit the `conanfile.py`:

- We are removing the `source` method because we have the sources in the same project; so we can use the `exports_sources`.
- In the `package_info` method, adjust the library name. In this case our `CMakeLists.txt` creates a target library called `mylibrary`.

12.2. IDEs
• Adjust the CMake helper in the build() method. The cmake.configure() doesn’t need to specify the source_folder, because we have the library.* files in the root directory.

• Adjust the copy function calls in the package method to ensure that all your headers and libraries are copied to the Conan package.

```python
from conans import ConanFile, CMake, tools

class MylibraryConan(ConanFile):
    name = "mylibrary"
    version = "1.0"
    license = "<Put the package license here>"
    url = "<Package recipe repository url here, for issues about the package>"
    description = "<Description of Mylibrary here>"
    settings = "os", "compiler", "build_type", "arch"
    options = {"shared": [True, False]}
    default_options = {"shared": False}
    generators = "cmake"
    requires = "zlib/1.2.11@conan/stable"

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

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

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

    def package_info(self):
        self.cpp_info.libs = ['mylibrary']
```

4. To build your library with CLion, follow the guide of *Using packages from step 5*.

5. To package your library, use the `conan export-pkg` command passing the used build-folder. It will call your package() method to extract the artifacts and push the Conan package to the local cache:

```
$ conan export-pkg . mylibrary/1.0@myuser/channel --build-folder cmake-build-debug --pr=myprofile
```

7. Now you can upload it to a Conan server if needed:

```
$ conan upload mylibrary/1.0@myuser/channel # This will upload only the recipe, use --all to upload all the generated binary packages.
```

8. If you would like to see how the package looks like before exporting it to the local cache (`conan export-pkg`) you can use the `conan package` command to create the package in a local directory:

```
$ conan package . --build-folder cmake-build-debug --package-folder=mypackage
```
If we list the `mypackage` folder we can see:

- A `lib` folder containing our library
- A `include` folder containing our header files
- A `conaninfo.txt` and `conanmanifest.txt` conan files, always present in all packages.

You can check a full example of a CLion project for creating a Conan package in this github repository: lasote/clion-conan-package.

12.2.3 Apple/Xcode

Conan can be integrated with Apple's XCode in two different ways:

- Using the `cmake` generator to create a `conanbuildinfo.cmake` file.
- Using the `xcode` generator to create a `conanbuildinfo.xcconfig` file.

With CMake

Check the Integrations/cmake section to read about the `cmake` generator. Check the official CMake docs to find out more about generating Xcode projects with CMake.

With the `xcode` generator

You can use the `xcode` generator to integrate your requirements with your Xcode project. This generator creates an `xcconfig` file, with all the `include paths, lib paths, libs, flags` etc, that can be imported in your project.

Open `conanfile.txt` and change (or add) the `xcode` generator:

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

[generators]
xcode
```

Install the requirements:

```
$ conan install .
```

Go to your Xcode project, click on the project and select Add files to...
Choose `conanbuildinfo.xcconfig` generated.

Click on the project again. In the `info/configurations` section, choose `conanbuildinfo` for `release` and `debug`.
Build your project as usual.

See also:
Check the Reference/Generators/xcode for the complete reference.

See also:
Check the Tools section about Apple tools to ease the integration with the Apple development tools in your recipes using the toolchain as a build require.

See also:
Check the Darwin Toolchain package section to learn how to cross build for iOS, watchOS and tvOS.

12.2.4 Android Studio

You can use Conan to cross-build your libraries for Android with different architectures. If you are using Android Studio for your Android application development, you can integrate Conan to automate the library building for the different architectures that you want to support in your project.

Here is an example of how to integrate the libpng Conan package library in an Android application, but any library that can be cross-compiled to Android could be used using the same procedure.

We are going to start from the “Hello World” wizard application and then will add it the libpng C library:

1. Follow the cross-build your libraries for Android guide to create a standalone toolchain and create a profile named android_21_arm_clang for Android. You can also use the NDK that the Android Studio installs.
2. Create a new Android Studio project and include C++ support.
3. Select your API level and target. The arch and api level have to match with the standalone toolchain created in step 1.

4. Add an empty Activity and name it.
5. Select the C++ standard
6. Change to the project view and in the app folder create a `conanfile.txt` with the following contents:

```
conanfile.txt

[requires]
libpng/1.6.23@lasote/stable

[generators]
cmake
```

7. Open the `CMakeLists.txt` file from the app folder and replace the contents with:

```
cmake_minimum_required(VERSION 3.4.1)
include(${CMAKE_CURRENT_SOURCE_DIR}/conan_build/conanbuildinfo.cmake)
set(CMAKE_CXX_COMPILER_VERSION "5.0") # Unknown miss-detection of the compiler by CMake
conan_basic_setup(TARGETS)
add_library(native-lib SHARED src/main/cpp/native-lib.cpp)
target_link_libraries(native-lib CONAN_PKG::libpng)
```

8. Open the `app/build.gradle` file. We are configuring the architectures we want to build, specifying adding a new task `conanInstall` that will call `conan install` to install the requirements:

- In the defaultConfig section, append:

```
ndk {
    // Specifies the ABI configurations of your native libraries Gradle should build and package with your APK.
    abiFilters 'armeabi-v7a'
}
```

- After the android block:
task conanInstall {
    def buildDir = new File("app/conan_build")
    buildDir.mkdirs()
    // if you have problems running the command try to specify the absolute
    // path to conan (Known problem in MacOSX) /usr/local/bin/conan
    def cmd = "conan install ../conanfile.txt --profile android_21_arm_clang --build
    →missing"
    print("$cmd")
    def sout = new StringBuilder(), serr = new StringBuilder()
    def proc = cmd.execute(null, buildDir)
    proc.consumeProcessOutput(sout, serr)
    proc.waitFor()
    println "$sout $serr"
    if(proc.exitValue() != 0){
        throw new Exception("out> $sout err> $serr" + "\nCommand: ${cmd}"")
    }
}

9. Finally open the default example cpp library in app/src/main/cpp/native-lib.cpp and include some
lines using your library. Be careful with the JNICALL name if you used another app name in the wizard:

```cpp
#include <jni.h>
#include <string>
#include "png.h"
#include "zlib.h"
#include <sstream>
#include <iostream>

extern "C"
JNIEXPORT jstring JNICALL
Java_com_jfrog_myconanandroidcppapp_MainActivity_stringFromJNI(
    JNIEnv *env,
    jobject /* this */) { 
    std::ostringstream oss;
    oss << "Compiled with libpng: " << PNG_LIBPNG_VER_STRING << std::endl;
    oss << "Running with libpng: " << png_libpng_ver << std::endl;
    oss << "Compiled with zlib: " << ZLIB_VERSION << std::endl;
    oss << "Running with zlib: " << zlib_version << std::endl;

    return env->NewStringUTF(oss.str().c_str());
}
```

Build your project normally. Conan will create a conan folder with a folder for each different architecture you have
specified in the abiFilters with a conanbuildinfo.cmake file.

Then run the app using an x86 emulator for best performance:
See also:
Check the section Linux/Windows/macOS to Android to read more about cross-building for Android.

12.2.5 YouCompleteMe (vim)

If you are a vim user, you may also be a user of YouCompleteMe.

With this generator, you can create the necessary files for your project dependencies, so YouCompleteMe will show symbols from your Conan installed dependencies for your project. You only have to add the `ycm` generator to your `conanfile`:

```
Listing 21: conanfile.txt

[generators]
ycm
```

It will generate a `conan_ycm_extra_conf.py` and a `conan_ycm_flags.json` file in your folder. Those files will be overwritten each time you run `conan install`.

In order to make YouCompleteMe work, copy/move `conan_ycm_extra_conf.py` to your project base folder (usually the one containing your `conanfile`) and rename it to `.ycm_extra_conf.py`.

You can (and probably should) edit this file to add your project specific configuration. If your base folder is different from your build folder, link the `conan_ycm_flags.json` from your build folder to your base folder.

```
# from your base folder
$ cp build/conan_ycm_extra_conf.py .ycm_extra_conf.py
$ ln -s build/conan_ycm_flags.json conan_ycm_flags.json
```
12.3 CI Platforms

You can use any CI platform to build your libraries and generate your Conan packages.

12.3.1 Jenkins

You can use Jenkins CI both for:

- Building and testing your project, which manages dependencies with Conan, and probably a conanfile.txt file
- Building and testing Conan binary packages for a given Conan package recipe (with a conanfile.py) and uploading to a Conan remote (Artifactory or conan_server)

There is no need for any special setup for it, just install Conan and your build tools in the Jenkins machine and call the needed Conan commands.

Artifactory and Jenkins integration

If you are using Artifactory you can take advantage of the Jenkins Artifactory Plugin. Check here how to install the plugin and here you can check the full documentation about the DSL.

The Artifactory Jenkins plugin provides a powerful DSL (Domain Specific Language) to call Conan, connect with your Artifactory instance, upload and download your packages from Artifactory and manage your build information.

Example: Test your project getting requirements from Artifactory

This is a template to use Jenkins with an Artifactory plugin and Conan to retrieve your package from Artifactory server and publish the build information about the downloaded packages to Artifactory.

In this script we assume that we already have all our dependencies in the Artifactory server, and we are building our project that uses **Boost** and **Poco** libraries.

Create a new Jenkins Pipeline task using this script:

```java
//Adjust your artifactory instance name/repository and your source code repository
def artifactory_name = "artifactory"
def artifactory_repo = "conan-local"
def repo_url = 'https://github.com/memsharded/example-boost-poco.git'
def repo_branch = 'master'

node {
    def server = Artifactory.server artifactory_name
def client = Artifactory.newConanClient()

    stage("Get project"){
        git branch: repo_branch, url: repo_url
    }
}
```

(continues on next page)
stage("Get dependencies and publish build info"){
    sh "mkdir -p build"
    dir ('build') {
        def b = client.run(command: "install ..")
        server.publishBuildInfo b
    }
}

stage("Build/Test project"){
    dir ('build') {
        sh "cmake ../ && cmake --build ."
    }
}
Example: Build a Conan package and upload it to Artifactory

In this example we will call Conan test package command to create a binary packages and then upload it to Artifactory. We also upload the build information:

```python
def artifactory_name = "artifactory"
def artifactory_repo = "conan-local"
def repo_url = 'https://github.com/conan-community/conan-zlib.git'
def repo_branch = "release/1.2.11"

node {
  def server = Artifactory.server artifactory_name
  def client = Artifactory.newConanClient()
  def serverName = client.remote.add server: server, repo: artifactory_repo

  stage("Get recipe") {
    git branch: repo_branch, url: repo_url
  }

  stage("Test recipe") {
    client.run(command: "create")
  }

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

Stage View

<table>
<thead>
<tr>
<th></th>
<th>Get recipe</th>
<th>Test recipe</th>
<th>Upload packages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average stage times:</td>
<td>797ms</td>
<td>47s</td>
<td>1s</td>
</tr>
<tr>
<td></td>
<td>587ms</td>
<td>1min 25s</td>
<td>1s</td>
</tr>
</tbody>
</table>

12.3. CI Platforms
12.3.2 Travis CI

You can use the Travis CI cloud service to automatically build and test your project in Linux/MacOS environments in the cloud. It is free for OSS projects, and offers an easy integration with GitHub, so builds can be automatically fired in Travis-CI after a `git push` to GitHub.

You can use Travis-CI both for:

- Building and testing your project, which manages dependencies with Conan, and probably a `conanfile.txt` file.
- Building and testing Conan binary packages for a given Conan package recipe (with a `conanfile.py`).

### Installing dependencies and building your project

A very common use case is to build your project after Conan takes care of installing your dependencies. Doing this process in Travis CI is quite convenient as you can do it with `conan install`.

To enable Travis CI support, you need to create a `.travis.yml` file and paste this code in it:

```yaml
os: linux
language: python
python: "3.7"
dist: xenial
compiler:
  - gcc
install:
  # Install conan
  - pip install conan
  # Automatic detection of your arch, compiler, etc.
  - conan user
script:
  # Download dependencies and build project
  - conan install .
  # Call your build system
  - cmake . -G "Unix Makefiles"
  - cmake --build .
  # Run your tests
  - ctest .
```

Travis will install the gcc compiler and the conan client and will execute the `conan install` command using the requirements and generators indicated in your `conanfile.py` or `conanfile.txt`. Then, the `script` section installs the requirements and then you can use your build system to compile the project (using `make` in this example).

### Creating, testing and uploading Conan binary packages

You can also use Travis CI to automate building new Conan binary packages with every change you push to GitHub. You can probably set up your own way, but Conan has some utilities to help in the process.

The command `conan new` has arguments to create a default working `.travis.yml` file. Other setups might be possible, but for this example we are assuming that you are using GitHub and also uploading your final packages to Bintray.
You could follow these steps:

1. First, create an empty GitHub repository. Let’s call it “hello”, for creating a “hello world” package. GitHub allows creating it with a Readme and .gitignore.

2. Get the credentials User and API Key. (Remember, Bintray uses the API key as “password”, not your main Bintray account password.)

3. Create a Conan repository in Bintray under your user or organization, and get its URL (“Set me up”). We will call it `UPLOAD_URL`.

4. Activate the repo in your Travis account, so it is built when we push changes to it.

5. Under Travis More Options -> Settings->Environment Variables, add the CONAN_PASSWORD environment variable with the Bintray API Key. If your Bintray user is different from the package user, you can also define your Bintray username, defining the environment variable `CONAN_LOGIN_USERNAME`.


7. Create the package: `conan new Hello/0.1@<user>/testing -t -s -cilg -cis -ciu=UPLOAD_URL` where `user` is your Bintray username.

8. You can inspect the created files: both `.travis.yml`, `.travis/run.sh`, and `.travis/install.sh` and the `build.py` script, that is used by `conan-package-tools` utility to split different builds with different configurations in different Travis CI jobs.

9. You can test locally, before pushing, with `conan test`.

10. Add the changes, commit and push: `git add . && git commit -m "first commit" && git push`.

11. Go to Travis and see the build, with the different jobs.

12. When it has finished, go to your Bintray repository, you should see there the uploaded packages for different configurations.

13. Check locally, searching in Bintray: `conan search Hello/0.1@<user>/testing -r=mybintray`.

If something fails, please report an issue in the `conan-package-tools` GitHub repository: https://github.com/conan-io/conan-package-tools

### 12.3.3 Appveyor

You can use the AppVeyor cloud service to automatically build and test your project in a Windows environment in the cloud. It is free for OSS projects, and offers an easy integration with Github, so builds can be automatically fired in Appveyor after a `git push` to Github.

You can use Appveyor both for:

- Building and testing your project, which manages dependencies with Conan, and probably a conanfile.txt file
- Building and testing Conan binary packages for a given Conan package recipe (with a conanfile.py)
Building and testing your project

We are going to use an example with GTest package, with AppVeyor support to run the tests.

Clone the project from github:

```
$ git clone https://github.com/lasote/conan-gtest-example
```

Create an `appveyor.yml` file and paste this code in it:

```yaml
version: 1.0.{build}
  platform:
    - x64

install:
  - cmd: echo "Downloading conan..."
  - cmd: set PATH=%PATH%;%PYTHON%/Scripts/
  - cmd: pip.exe install conan
  - cmd: conan user # Create the conan data directory
  - cmd: conan --version

build_script:
  - cmd: mkdir build
  - cmd: conan install . -o gtest:shared=True
  - cmd: cd build
  - cmd: cmake .. / -DBUILD_TEST=TRUE -G "Visual Studio 14 2015 Win64"
  - cmd: cmake --build . --config Release

test_script:
  - cmd: cd bin
  - cmd: encryption_test.exe
```

Appveyor will install the Conan tool and will execute the `conan install` command. Then, the `build_script` section creates the build folder, compiles the project with `cmake` and the section `test_script` runs the tests.

Creating, testing and uploading Conan binary packages

You can use Appveyor to automate the building of binary packages, which will be created in the cloud after pushing to Github. You can probably set up your own way, but Conan has some utilities to help in the process.

The command `conan new` has arguments to create a default working `appveyor.yml` file. Other setups might be possible, but for this example we are assuming that you are using GitHub and also uploading your final packages to Bintray. You could follow these steps:

1. First, create an empty github repository. Let’s call it “hello”, for creating a “hello world” package. Github allows to create it with a Readme and .gitignore.
2. Get the credentials User and API Key. (Remember, Bintray uses the API key as “password”, not your main Bintray account password.)
3. Create a Conan repository in Bintray under your user or organization, and get its URL (“Set me up”). We will call it `UPLOAD_URL`
4. Activate the repo in your Appveyor account, so it is built when we push changes to it.
5. Under Appveyor Settings->Environment, add the `CONAN_PASSWORD` environment variable with the Bintray API Key, and encrypt it. If your Bintray user is different from the package user, you can define your Bintray username too, defining the environment variable `CONAN_LOGIN_USERNAME`
6. Clone the repo: 
   `$ git clone <your_repo/hello> && cd hello`

7. Create the package: 
   `conan new Hello/0.1@<user>/testing -t -s -ciw -cis `ciu=UPLOAD_URL` where `user` is your Bintray username

8. You can inspect the created files: both `appveyor.yml` and the `build.py` script, that is used by `conan-package-tools` utility to split different builds with different configurations in different appveyor jobs.

9. You can test locally, before pushing, with `conan create`

10. Add the changes, commit and push: 
    `git add . && git commit -m "first commit" && git push`

11. Go to Appveyor and see the build, with the different jobs.

12. When it finish, go to your Bintray repository, you should see there the uploaded packages for different configurations.

13. Check locally, searching in Bintray: 
    `conan search Hello/0.1@<user>/testing -r=mybintray`

If something fails, please report an issue in the `conan-package-tools` github repository: 
[https://github.com/conan-io/conan-package-tools](https://github.com/conan-io/conan-package-tools)

### 12.3.4 Gitlab

You can use the Gitlab CI cloud or local service to automatically build and test your project in Linux/MacOS/Windows environments. It is free for OSS projects, and offers an easy integration with Gitlab, so builds can be automatically fired in Gitlab CI after a `git push` to Gitlab.

You can use Gitlab CI both for:

- Building and testing your project, which manages dependencies with Conan, and probably a conanfile.txt file
- Building and testing Conan binary packages for a given Conan package recipe (with a conanfile.py)

**Building and testing your project**

We are going to use an example with GTest package, with Gitlab CI support to run the tests.

Clone the project from github:

```
$ git clone https://github.com/lasote/conan-gtest-example
```

Create a `.gitlab-ci.yml` file and paste this code in it:

```yaml
image: conanio/gcc63

build:
  before_script:
    # Upgrade Conan version
    - sudo pip install --upgrade conan
    # Automatic detection of your arch, compiler, etc.
    - conan user
```

(continues on next page)
Gitlab CI will install the conan tool and will execute the conan install command. Then, the script section creates the build folder, compiles the project with cmake and runs the tests.

On Windows the Gitlab runner may be running as a service and not have a home directory, in which case you need to set a custom value for CONAN_USER_HOME.

Creating, testing and uploading Conan binary packages

You can use Gitlab CI to automate the building of binary packages, which will be created in the cloud after pushing to Gitlab. You can probably setup your own way, but Conan has some utilities to help in the process. The command conan new has arguments to create a default working .gitlab-ci.yml file. Other setups might be possible, but for this example we are assuming that you are using github and also uploading your final packages to Bintray. You could follow these steps:

1. First, create an empty gitlab repository, let’s call it “hello”, for creating a “hello world” package. Gitlab allows to create it with a Readme, license and .gitignore.
2. Get the credentials User and API Key (remember, Bintray uses the API key as “password”, not your main Bintray account password)
3. Create a Conan repository in Bintray under your user or organization, and get its URL (“Set me up”). We will call it UPLOAD_URL
4. Under your project page, Settings -> Pipelines -> Add a variable, add the CONAN_PASSWORD environment variable with the Bintray API Key. If your Bintray user is different from the package user, you can also define your Bintray username, defining the environment variable CONAN_LOGIN_USERNAME
5. Clone the repo: git clone <your_repo/hello> && cd hello.
6. Create the package: conan new Hello/0.1@<user>/testing -t -s -ciglg -ciglc -cis -ciu=UPLOAD_URL where user is your Bintray username.
7. You can inspect the created files: both .gitlab-ci.yml and the build.py script, that is used by conan-package-tools utility to split different builds with different configurations in different GitLab CI jobs.
8. You can test locally, before pushing, with conan create or by GitLab Runner.
9. Add the changes, commit and push: git add . && git commit -m "first commit" && git push.
10. Go to Pipelines page and see the pipeline, with the different jobs.
11. When it has finished, go to your Bintray repository, you should see there the uploaded packages for different configurations.
12. Check locally, searching in Bintray: conan search Hello/0.1@<user>/testing -r=mybintray.

If something fails, please report an issue in the conan-package-tools github repository: https://github.com/conan-io/conan-package-tools
12.3.5 Circle CI

You can use the Circle CI cloud to automatically build and test your project in Linux/MacOS environments. It is free for OSS projects, and offers an easy integration with Github, so builds can be automatically fired in CircleCI after a git push to Github.

You can use CircleCI both for:

- Building and testing your project, which manages dependencies with Conan, and probably a conanfile.txt file
- Building and testing Conan binary packages for a given Conan package recipe (with a conanfile.py)

Building and testing your project

We are going to use an example with GTest package, with CircleCI support to run the tests.

Clone the project from github:

```
$ git clone https://github.com/lasote/conan-gtest-example
```

Create a `.circleci/config.yml` file and paste this code in it:

```yaml
version: 2
gcc-6:
  docker:
    - image: conanio/gcc6
  steps:
    - checkout
    - run:
      name: Build Conan package
      command: |
        sudo pip install --upgrade conan
        conan user
        conan create . user/channel
  workflows:
    version: 2
    build_and_test:
      jobs:
        - gcc-6
```

CircleCI will install the Conan tool and will execute the `conan create` command. Then, the `script` section creates the build folder, compiles the project with `cmake` and runs the tests.

Creating, testing and uploading Conan package binaries

You can use CircleCI to automate the building of binary packages, which will be created in the cloud after pushing to Github. You can probably set up your own way, but Conan has some utilities to help in the process.

The command `conan new` has arguments to create a default working `.circleci/config.yml` file. Other setups might be possible, but for this example we are assuming that you are using github and also uploading your final packages to Bintray. You could follow these steps:
1. First, create an empty Github repository (let’s call it “hello”) for creating a “hello world” package. Github allows to create it with a Readme, license and .gitignore.

2. Get the credentials User and API Key (remember, Bintray uses the API key as “password”, not your main Bintray account password)

3. Create a Conan repository in Bintray under your user or organization, and get its URL (“Set me up”). We will call it UPLOAD_URL

4. Under your project page, Settings -> Pipelines -> Add a variable, add the CONAN_PASSWORD environment variable with the Bintray API Key. If your Bintray user is different from the package user, you can also define your Bintray username, defining the environment variable CONAN_LOGIN_USERNAME

5. Clone the repo: $ git clone <your_repo/hello> && cd hello

6. Create the package:  $ conan new Hello/0.1@<user>/testing -t -s -ciccg -ciccc -cicco -cis -ciu=UPLOAD_URL where user is your Bintray username

7. You can inspect the created files: both .circleci/config.yml and the build.py script, that is used by conan-package-tools utility to split different builds with different configurations in different GitLab CI jobs.

8. You can test locally, before pushing, with $ conan create

9. Add the changes, commit and push: $ git add . && git commit -m "first commit" && git push

10. Go to Pipelines page and see the pipeline, with the different jobs.

11. When it has finished, go to your Bintray repository, you should see there the uploaded packages for different configurations

12. Check locally, searching in Bintray: $ conan search Hello/0.1@<user>/testing -r=mybintray

If something fails, please report an issue in the conan-package-tools github repository: https://github.com/conan-io/conan-package-tools

12.3.6 Microsoft’s Azure DevOps (TFS, VSTS)

Thanks to the JFrog Artifactory Extension for Azure DevOps and TFS it is possible to support Conan tasks and integrate it with the CI development platform provided by Microsoft’s Azure DevOps and the Artifactory binary repository manager.

The support for Conan now in the JFrog Artifactory Extension helps you perform the following tasks in Azure DevOps or TFS:

• Run Conan commands
• Resolve Conan dependencies from remote Artifactory servers
• Push Conan packages to Artifactory
• Publish BuildInfo metadata
• Import a Conan configuration

In this section we will show you how to add Conan tasks to your pipelines using the Artifactory/Conan Extension and push the generated buildinfo metadata to Artifactory where it can be used to track and automate your builds.
Configuring DevOps Azure to use Artifactory with Conan

To use the Conan support provided by the JFrog Artifactory Extension you must configure a self-hosted agent that will enable Conan builds for your Azure Pipelines environment. Afterwards you can install the JFrog Artifactory Extension from the Visual Studio Marketplace and follow the installation instructions in the Overview.

When completed, proceed to create builds and access buildinfo from within Azure DevOps or TFS.

Steps to follow

In these steps, you will set up Azure DevOps to use Artifactory and add Conan tasks to your build pipeline. Then you can set up to push the buildinfo from the Conan task to Artifactory.
STEP 1: Configure the Artifactory instance

Once the Artifactory Extension is installed, you must configure Azure DevOps to access the Artifactory instance.

To add Artifactory to Azure DevOps:

1. In Azure DevOps, go to Project Settings > Service connections.
2. Click + New service connection to display the list control, and select Artifactory.
3. In the resulting Update Authentication for Artifactory dialog, enter the required server and credential information, and click OK.
STEP 2: Add a Conan task

Once your Artifactory connection is configured, you may add Conan tasks to your Build or Release pipelines.

To add a Conan task:

1. Go to the Pipeline Tasks setup screen.
2. In the Add tasks section, search for “Conan” in the task selection list.
3. Select the Artifactory Conan task to add it to your pipeline.
4. In the new task, select which Conan command to run.

5. Configure the Conan command for the task.
Continue to add Conan tasks as you need for each pipeline.

**STEP 3: Configure the Push task buildinfo to Artifactory**

When the pipeline containing the Conan task executes, the task log shows all the information about the executed Conan command.
You can configure your Conan task to collect the buildinfo by selecting the Collect buildinfo checkbox when you create the task.
Once collected, the buildinfo can then be pushed as metadata to Artifactory.

To perform this, create an Artifactory Publish Build Info task to push the metadata to your Artifactory instance.
After you run the pipeline, you will be able to see the build information for the Conan task in Artifactory.
12.4 Other Systems

You can run Conan on any platform supporting Python and also cross-build Conan packages for different platforms.

12.4.1 Docker

You can easily run Conan in a Docker container to build and cross-build conan packages. Check the ‘How to use docker to create and cross build C and C++ conan packages’ section to know more.

See also:
The documentation for this integration is taken from the JFrog blog.
12.4.2 Emscripten

It should be possible to build packages for Emscripten (asm.js) via the following conan profile:

```conan
include(default)
[settings]
os=Emscripten
arch=asm.js
compiler=clang
compiler.version=6.0
compiler.libcxx=libc++

[options]
[build_requires]
emsdk_installer/1.38.29@bincrafters/stable

[env]
```

And the following conan profile is required for the WASM (Web Assembly):

```conan
include(default)
[settings]
os=Emscripten
arch=wasm
compiler=clang
compiler.version=6.0
compiler.libcxx=libc++

[options]
[build_requires]
emsdk_installer/1.38.29@bincrafters/stable

[env]
```

These profile above are using the emsdk_installer/1.38.29@bincrafters/stable conan package. It will automatically download the Emscripten SDK and set up required environment variables (like CC, CXX, etc.).

**Note:** In order to use emsdk_installer package, you need to add it to the remotes:

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

**Note:** Alternatively, it’s always possible to use an existing emsdk installation and manually specify required environment variables within the [env] section of the conan profile.

**Note:** In addition to the above, Windows users may need to specify CONAN_MAKE_PROGRAM, for instance from the existing MinGW installation (e.g. C:\MinGW\bin\mingw32-make.exe), or use make from the mingw_installer/1.0@conan/stable.

**Note:** In addition to the above, Windows users may need to specify CONAN_CMAKE_GENERATOR, e.g. to MinGW Makefiles, because default one is Visual Studio. Other options (e.g. Ninja) work as well.
As specified, os has been set to the Emscripten, and arch has been set to either asm.js or wasm (only these two are currently supported). And compiler setting has been set to match the one used by Emscripten - Clang 6.0 with libc++ standard library.

### Running the code inside the browser

**Note:** Emscripten requires Python 2.7.12 or above, make sure that you have an up-to-date Python version installed.

**Note:** Running demo on Windows may require pywin32 module. Install it by running `pip install pywin32`.

In order to demonstrate how to use conan with Emscripten, let's check out the example project:

```
$ git clone --depth 1 git@github.com:conan-io/examples.git
```

Change the directory to the Emscripten demo:

```
$ cd features
$ cd emscripten
```

This is an extremely simple demo, which just imports the famous zlib library and outputs its version into the browser.

In order to build it for the Emscripten run:

```
$ ./build.sh
```

or (on Windows):

```
$ ./build.cmd
```

Please note that running the above command may take a while to download and build required dependencies. This script will execute several conan commands:

```
$ conan remove conan-hello-emscripten/* -f
$ conan create . conan/testing -k -p emscripten.profile --build missing
$ conan install conanfile.txt -pr emscripten.profile
```

First one removes any traces of previous demo installations, just to ensure that environment is clean. Then, it builds the simple demo (it uses CMakeLists.txt and main.cpp files from the current directory). The following local profile is used (file emscripten.profile within the current directory):

```ini
include(default)
[settings]
os=Emscripten
arch=wasm
compiler=clang
compiler.version=6.0
compiler.libcxx=libc++
[options]
[buildRequires]
emsdk_installer/1.38.29@bincrafters/stable
ninja_installer/1.8.2@bincrafters/stable
[env]
```

12.4. Other Systems
Finally, it installs the demo importing the required files (.html, .js and .wasm) into the bin subdirectory.

Then we can run the code inside the browser via emrun helper:

```bash
$ ./run.sh
```

or (on Windows):

```bash
$ ./run.cmd
```

The command above uses virtualenv generator generator in order to get emrun command available in the PATH. And as the result, Web Browser should be opened (or new tab in Web Browser will be opened, if it was already run), and the following output should be displayed:

```bash
$ Using zlib version: 1.2.11
```

It confirms the fact we have just built zlib into JavaScript and run it inside the Web Browser.

## 12.4.3 QNX SOFTWARE SYSTEMS QNX Neutrino

It’s possible to cross-compile packages for QNX Neutrino operation system with conan. Conan has support for QNX Neutrino 6.x and 7.x. The following architectures are supported:

- armv7
- armv8
- sh4le
- ppc32be

The following C++ standard library implementations are supported for QCC:

- cxx (LLVM C++)
- gpp (GNU C++)
- cpp (Dinkum C++)
- cpp-ne (Dinkum C++ without exceptions)
- acpp (Dinkum Abridged C++)
- acpp-ne (Dinkum Abridged C++ without exceptions)
- ecpp (Dinkum Embedded C++)
- ecpp-ne (Dinkum Embedded C++ without exceptions)

Conan automatically sets up corresponding compiler flags for the given standard library (e.g. `-Y cxx` for the LLVM C++).

With QNX SDK set up on the machine, the following conan profile might be used for the cross-compiling (assuming qcc in the PATH):
12.5 Version Control System

Conan uses plain text files for the recipes and configuration files and they can be managed nicely with any version control system. Also, with the `scm` feature, your recipe can capture automatically the commit/revision of the source code of your library so the recipe will clone the correct sources automatically.

12.5.1 Git

Conan uses plain text files, `conanfile.txt` or `conanfile.py`, so it’s perfectly suitable for the use of any version control system. We use and highly recommend `git`. Check workflows section to learn more about project layouts that naturally fit version control systems.

Temporary files

Conan generates some files that should not be committed, as `conanbuildinfo.*` and `conaninfo.txt`. These files can change in different computers and are re-generated with the `conan install` command.

However, these files are typically generated in the build tree not in the source tree, so they will be naturally disregarded. Just take care in case you have created the build folder inside your project (we do this in several examples in the documentation). In this case, you should add it to your `.gitignore` file:

```
Listing 22: .gitignore
...
build/
```

Package creators

Check `scm feature` to learn more about managing the libraries source code with Git.

If you are creating a Conan package:

- You can use the `url` field to indicate the origin of your package recipe. If you are using an external package recipe, this url should point to the package recipe repository not to the external source origin. If a `github` repository is detected, the Conan website will link your github issues page from your Conan’s package page.

- You can use `git` to obtain your source (requires the git client in the path) when creating external package recipes.
12.5.2 SVN

Conan uses plain text files, `conanfile.txt` or `conanfile.py`, so it’s perfectly suitable for the use of any version control system.

Check [workflows section](#) to learn more about project layouts that naturally fit version control systems.

Check [scm feature](#) to learn more about managing the libraries source code with SVN.

12.6 Custom integrations

If you intend to use a build system that does not have a built-in generator, you may still be able to do so. There are several options:

- First, search in Bintray for generator packages. Generators can be created and contributed by users as regular packages, so you can depend on them as a normal requirement, use versioning and evolve faster without depending on the Conan releases.
- You can use the `txt` or `json` generators. They will generate a text file, simple to read that you can easily parse with your tools to extract the required information.
- Use the `conanfile data model` (`deps_cpp_info`, `deps_env_info`) in your recipe to access its properties and values, so you can directly call your build system with that information, without requiring to generate a file.
- Write and [create your own generator](#). So you can upload it, version and reuse it, as well as share it with your team or community. Check [How to create and share a custom generator with generator packages](#).

Note: Need help integrating your build system? Tell us what you need: info@conan.io

12.6.1 Use the JSON generator

Specify the `json` generator in your recipe:

Listing 23: `conanfile.txt`

```ini
[requires]
fmt/5.3.0@bincrafters/stable
Poco/1.9.0@pocoproject/stable

[generators]
json
```

A file named `conanbuildinfo.json` will be generated. It will contain the information about every dependency:
Listing 24: `conanbuildinfo.json`

```json
{
  "dependencies": [
    {
      "name": "fmt",
      "version": "5.3.0",
      "include_paths": [
        "/path/to/.conan/data/fmt/5.3.0/bincrafters/stable/package/<id>/include"
      ],
      "lib_paths": [
        "/path/to/.conan/data/fmt/5.3.0/bincrafters/stable/package/<id>/lib"
      ],
      "libs": [
        "fmt"
      ],
      "...": "...",
    },
    {
      "name": "Poco",
      "version": "1.9.0",
      "...": "...	"
    }
  ]
}
```

### 12.6.2 Use the text generator

Just specify the `txt` generator in your recipe:

Listing 25: `conanfile.txt`

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

[generators]
txt
```

A file is generated with the same information in a generic text format.

Listing 26: `conanbuildinfo.txt`

```
[includedirs]
/home/laso/.conan/data/Poco/1.6.1/lasote/stable/package/
  →aafac631e705f7296bec38318b28e4361ab6787c/include
/home/laso/.conan/data/OpenSSL/1.0.2d/lasote/stable/package/
  →154942d8bccc87fbbaa9157edae62e1200e80fc/include
/home/laso/.conan/data/zlib/1.2.8/lasote/stable/package/
  →3b92a20cb586af0d984797002d12b7120d38e95e/include

[libs]
PocoUtil
PocoXML
PocoJSON
PocoMongoDB
```

(continues on next page)
[bindirs]
/home/laso/.conan/data/Poco/1.6.1/lasote/stable/package/
→afafc631e705f7296bec38318b28e4361ab6787c/lib
/home/laso/.conan/data/OpenSSL/1.0.2d/lasote/stable/package/
→154942d8b7cc87fbb91957eldae9e262e1200e800fbc/lib
/home/laso/.conan/data/zlib/1.2.8/lasote/stable/package/
→3b92a20cb586af0d984797002d12b7120d38e95e/lib

[defines]
POCO_STATIC=ON
POCO_NO_AUTOMATIC_LIBS

[USER_MyRequiredLib1]
somevariable=Some Value
othervar=Othervalue

[USER_MyRequiredLib2]
myvar=34

12.6.3 Use the Conan data model (in a conanfile.py)

If you are using any other build system you can use Conan too. In the build() method you can access your settings and build information from your requirements and pass it to your build system. Note, however, that probably is simpler and much more reusable to create a generator to simplify the task for your build system.

Listing 27: conanfile.py

```python
from conans import ConanFile

class MyProjectWithConan(ConanFile):
    settings = "os", "compiler", "build_type", "arch"
    requires = "Poco/1.9.0@pocoproject/stable"
    
    """
    IT'S IMPORTANT TO DECLARE THE TXT GENERATOR TO DEAL WITH A GENERIC BUILD SYSTEM
    """

    # BUILD SYSTEM
```

(continues on next page)
generators = "txt"
default_options = {"Poco:shared": False, "OpenSSL:shared": False}

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

def build(self):
    ############ Without any helper ############
    # Settings
    print(self.settings.os)
    print(self.settings.arch)
    print(self.settings.compiler)

    # Options
    #print(self.options.my_option)
    print(self.options["OpenSSL"].shared)
    print(self.options["Poco"].shared)

    # Paths and libraries, all
    print("-------- ALL --------------")
    print(self.deps_cpp_info.include_paths)
    print(self.deps_cpp_info.lib_paths)
    print(self.deps_cpp_info.bin_paths)
    print(self.deps_cpp_info.libs)
    print(self.deps_cpp_info.defines)
    print(self.deps_cpp_info.cflags)
    print(self.deps_cpp_info.cxxflags)
    print(self.deps_cpp_info.sharedlinkflags)
    print(self.deps_cpp_info.exelinkflags)

    # Just from OpenSSL
    print("-------- FROM OPENSSL ------------")
    print(self.deps_cpp_info["OpenSSL"].include_paths)
    print(self.deps_cpp_info["OpenSSL"].lib_paths)
    print(self.deps_cpp_info["OpenSSL"].bin_paths)
    print(self.deps_cpp_info["OpenSSL"].libs)
    print(self.deps_cpp_info["OpenSSL"].defines)
    print(self.deps_cpp_info["OpenSSL"].cflags)
    print(self.deps_cpp_info["OpenSSL"].cxxflags)
    print(self.deps_cpp_info["OpenSSL"].sharedlinkflags)
    print(self.deps_cpp_info["OpenSSL"].exelinkflags)

    # Just from POCO
    print("-------- FROM POCO ------------")
    print(self.deps_cpp_info["Poco"].include_paths)
    print(self.deps_cpp_info["Poco"].lib_paths)
    print(self.deps_cpp_info["Poco"].bin_paths)
    print(self.deps_cpp_info["Poco"].libs)
    print(self.deps_cpp_info["Poco"].defines)
    print(self.deps_cpp_info["Poco"].cflags)
    print(self.deps_cpp_info["Poco"].cxxflags)
    print(self.deps_cpp_info["Poco"].sharedlinkflags)
    print(self.deps_cpp_info["Poco"].exelinkflags)

    # self.run("invoke here your configure, make, or others")
    # self.run("basically you can do what you want with your requirements build_
# Environment variables (from requirements self.env_info objects) # are automatically applied in the python `os.environ` but can be accesible as well:
```python
print("-------- Globally --------------")
print(self.env)

print("-------- FROM MyLib -----------")
print(self.deps_env_info["MyLib"].some_env_var)
```

# User declared variables (from requirements self.user_info objects) # are available in the self.deps_user_info object
```python
print("-------- FROM MyLib -----------")
print(self.deps_user_info["MyLib"].some_user_var)
```

## 12.6.4 Create your own generator

There are two ways in which generators can be contributed:

- Forking and adding the new generator in the Conan codebase. This will be a built-in generator. It might have a much slower release and update cycle, it needs to pass some tests before being accepted, but it has the advantage than no extra things are needed to use that generator (once next Conan version is released).

- Creating a custom generator package. You can write a `conanfile.py` and add the custom logic for a generator inside that file, then upload, refer and depend on it as any other package. These generators will be another node in the dependency graph but they have many advantages: much faster release cycles, independent from the Conan codebase and can be versioned. So backwards compatibility and upgrades are much easier.

## 12.6.5 Extending Conan

There are other powerful mechanisms to integrate other tools with Conan. Check the *Extending Conan* section for further information.

## 12.7 Linting

You can develop your recipe and binary packages getting feedback of potential issues.

### 12.7.1 Linting the recipe

**conanfile.py**

The `conan create` command verifies the recipe file using pylint.

However, if you have an IDE that supports Python and may do linting automatically, there are false warnings caused by the fact that Conan dynamically populates some fields of the recipe based on context.

Conan provides a plugin which makes pylint aware of these dynamic fields and their types. To use it when running pylint outside Conan, just add the following to your `.pylintrc` file:

```ini
[MASTER]
load-plugins=conans.pylint_plugin
```
Custom rules

Using the Conan hooks feature you can perform sanity checks to your recipe.
Take a look at the official hooks repository to see several examples of how to implement a linter for your recipe.

12.7.2 Linting binary packages

Using the Conan hooks feature you can scan your binaries to ensure that you are generating the correct binary files and even checking the binary contents.
Take a look at the official hooks repository to see several examples of how to implement a binary linter system.
This section shows common solutions and different approaches to typical problems.

13.1 How to package header-only libraries

13.1.1 Without unit tests

Packaging a header only library, without requiring to build and run unit tests for it within Conan, can be done with a very simple recipe. Assuming you have the recipe in the source repo root folder, and the headers in a subfolder called include, you could do:

```python
from conans import ConanFile

class HelloConan(ConanFile):
    name = "Hello"
    version = "0.1"
    # No settings/options are necessary, this is header only
    exports_sources = "include/*"
    no_copy_source = True

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

If you want to package an external repository, you can use the `source()` method to do a clone or download instead of the `exports_sources` fields.

- There is no need for `settings`, as changing them will not affect the final package artifacts
- There is no need for `build()` method, as header-only are not built
- There is no need for a custom `package_info()` method. The default one already adds an “include” subfolder to the include path
- `no_copy_source = True` will disable the copy of the source folder to the build directory as there is no need to do so because source code is not modified at all by the `configure()` or `build()` methods.
- Note that this recipe has no other dependencies, settings or options. If it had any of those, it would be very convenient to add the `package_id()` method, to ensure that only one package with always the same ID is created, irrespective of the configurations and dependencies:

```python
def package_id(self):
    self.info.header_only()
```

Package is created with:
13.1.2 With unit tests

If you want to run the library unit test while packaging, you would need this recipe:

```python
from conans import ConanFile, CMake

class HelloConan(ConanFile):
    name = "Hello"
    version = "0.1"
    settings = "os", "compiler", "arch", "build_type"
    exports_sources = "include/*", "CMakeLists.txt", "example.cpp"
    no_copy_source = True

    def build(self):  # this is not building a library, just tests
        cmake = CMake(self)
        cmake.configure()
        cmake.build()
        cmake.test()

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

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

Tip: If you are cross-building your library or app you’ll probably need to skip the unit tests because your target binary cannot be executed in current building host. To do it you can use `tools.get_env()` in combination with `CONAN_RUN_TESTS` environment variable, defined as False in profile for cross-building and replace `cmake.test()` with:

```python
if tools.get_env("CONAN_RUN_TESTS", True):
    cmake.test()
```

Which will use a `CMakeLists.txt` file in the root folder:

```cpp
project(Package CXX)
cmake_minimum_required(VERSION 2.8.12)
include_directories("include")
add_executable(example example.cpp)
enable_testing()
add_test(NAME example
        WORKING_DIRECTORY ${CMAKE_BINARY_DIR}/bin
        COMMAND example)
```

and some `example.cpp` file, which will be our “unit test” of the library:

```cpp
#include <iostream>
#include "hello.h"
```
int main() {
    hello();
}

- This will use different compilers and versions, as configured by Conan settings (in command line or profiles), but will always generate just 1 output package, always with the same ID.
- The necessary files for the unit tests, must be exports_sources too (or retrieved from source() method)
- If the package had dependencies, via requires, it would be necessary to add the generators = "cmake" to the package recipe and adding the conanbuildinfo.cmake file to the testing CMakeLists.txt:

```c
include(${CMAKE_BINARY_DIR}/conanbuildinfo.cmake)
conan_basic_setup()
add_executable(example example.cpp)
target_link_libraries(example ${CONAN_LIBS}) # not necessary if dependencies are also header-only
```

Package is created with:

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

**Note:** This with/without tests is referring to running full unitary tests over the library, which is different to the test functionality that checks the integrity of the package. The above examples are describing the approaches for unit-testing the library within the recipe. In either case, it is recommended to have a test_package folder, so the conan create command checks the package once it is created. Check the packaging getting started guide

### 13.2 How to launch conan install from cmake

It is possible to launch conan install from cmake, which can be convenient for end users, package consumers, that are not creating packages themselves.

This is work under testing. Please try it and give feedback or contribute. The CMake code to do this task is here:

[https://github.com/conan-io/cmake-conan](https://github.com/conan-io/cmake-conan)

To be able to use it, you can directly download the code from your CMake script:

**Listing 1: CMakeLists.txt**

```cmake
cmake_minimum_required(VERSION 2.8)
project(myproject CXX)

# Download automatically, you can also just copy the conan.cmake file
if(NOT EXISTS "${CMAKE_BINARY_DIR}/conan.cmake")
    message(STATUS "Downloading conan.cmake from https://github.com/conan-io/cmake-
              conan")
               cmake" "${CMAKE_BINARY_DIR}/conan.cmake")
endif()

include("${CMAKE_BINARY_DIR}/conan.cmake")
```

(continues on next page)
conan Documentation, Release 1.16.0

(concluded from previous page)

```conan
conan_cmake_run(REQUIRES Catch2/2.6.0@catchorg/stable
    BASIC_SETUP)
add_executable(main main.cpp)
target_link_libraries(main ${CONAN_LIBS})
```

If you want to use targets, you could do:

```conan
include(conan.cmake)
conan_cmake_run(REQUIRES Catch2/2.6.0@catchorg/stable
    BASIC_SETUP CMAKE_TARGETS)
add_executable(main main.cpp)
target_link_libraries(main CONAN_PKG::Hello)
```

13.3 How to create and reuse packages based on Visual Studio

Conan has different helpers to manage Visual Studio and MSBuild based projects. This how-to illustrates how to put them together to create and consume packages that are purely based on Visual Studio. This how-to is using VS2015, but other versions can be used too.

13.3.1 Creating packages

Start cloning the existing example repository, containing a simple “Hello World” library, and application:

```bash
$ git clone https://github.com/memsharded/hello_vs
$ cd hello_vs
```

It contains a src folder with the source code and a build folder with a Visual Studio 2015 solution, containing 2 projects: the HelloLib static library, and the Greet application. Open it:

```bash
$ build\HelloLib\HelloLib.sln
```

You should be able to select the Greet subproject -> Set as Startup Project. Then build and run the app with Ctrl+F5. (Debug -> Start Without Debugging)

```bash
$ Hello World Debug!
# Switch IDE to Release mode, repeat
$ Hello World Release!
```

Because the hello.cpp file contains an `#ifdef _DEBUG` to switch between debug and release message.

In the repository, there is already a conanfile.py recipe:

```python
from conans import ConanFile, MSBuild
class HelloConan(ConanFile):
    name = "Hello"
    version = "0.1"
    license = "MIT"
    url = "https://github.com/memsharded/hello_vs"
```

(continues on next page)
settings = "os", "compiler", "build_type", "arch"
exports_sources = "src/*", "build/*"

def build(self):
    msbuild = MSBuild(self)
    msbuild.build("build/HelloLib/HelloLib.sln")

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

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

This recipe is using the MSBuild() build helper to build the sln project. If our recipe has requires, the MSBUILD helper will also take care of inject all the needed information from the requirements, as include directories, library names, definitions, flags etc to allow our project to locate the declared dependencies.

The recipe contains also a test_package folder with a simple example consuming application. In this example, the consuming application is using cmake to build, but it could also use Visual Studio too. We have left the cmake one because it is the default generated with conan new, and also to show that packages created from Visual Studio projects can also be consumed with other build systems like CMake.

Once we want to create a package, it is advised to close VS IDE, clean the temporary build files from VS to avoid problems, then create and test the package. Here it is using system defaults, assuming they are Visual Studio 14, Release, x86_64:

```
# close VS
$ git clean -xdf
$ conan create . memsharded/testing ...
> Hello World Release!
```

Instead of closing the IDE and running the command:git clean we could also configure a smarter filter in exports_sources field, so temporary build files are not exported into the recipe.

This process can be repeated to create and test packages for different configurations:

```
$ conan create . memsharded/testing -s arch=x86
$ conan create . memsharded/testing -s compiler="Visual Studio" -s compiler.runtime=MDd -s build_type=Debug
$ conan create . memsharded/testing -s compiler="Visual Studio" -s compiler.runtime=MDd -s arch=x86
```

**Note:** It is not mandatory to specify the compiler.runtime setting. If it is not explicitly defined, Conan will automatically use runtime=MDd for build_type=Debug and runtime=MD for build_type=Release.

You can list the different created binary packages:

```
$ conan search Hello/0.1@memsharded/testing
```
13.3.2 Uploading binaries

Your locally created packages can already be uploaded to a Conan remote. If you created them with the original username “memsharded”, as from the git clone, you might want to do a **conan copy** to put them on your own username. Of course, you can also directly use your user name in **conan create**.

Another alternative is to configure the permissions in the remote, to allow uploading packages with different usernames. By default artifactory will do it but Conan server won’t: permissions must be given in **[write_permissions]** section of **server.conf**.

13.3.3 Reusing packages

To use existing packages directly from Visual Studio, Conan provides the **visual_studio** generator. Let’s clone an existing “Chat” project, consisting of a ChatLib static library that makes use of the previous “Hello World” package, and a MyChat application, calling the ChatLib library function.

```bash
$ git clone https://github.com/memsharded/chat_vs
$ cd chat_vs
```

As above, the repository contains a Visual Studio solution in the **build** folder. But if you try to open it, it will fail to load. This is because it is expecting to find a file with the required information about dependencies, so it is necessary to obtain that file first. Just run:

```
$ conan install .
```

You will see that it created two files, a **conaninfo.txt** file, containing the current configuration of dependencies, and a **conanbuildinfo.props** file, containing the Visual Studio properties (like `<AdditionalIncludeDirectories>`), so it is able to find the installed dependencies.

Now you can open the IDE and build and run the app (by the way, the chat function is just calling the **hello()** function two or three times, depending on the build type):

```
$ build\ChatLib\ChatLib.sln
# Switch to Release
# MyChat -> Set as Startup Project
# Ctrl + F5 (Debug -> Run without debugging)
> Hello World Release!
> Hello World Release!
```

If you wish to link with the debug version of Hello package, just install it and change IDE build type:

```
$ conan install . -s build_type=Debug -s compiler="Visual Studio" -s compiler.runtime=MDd
# Switch to Debug
# Ctrl + F5 (Debug -> Run without debugging)
> Hello World Debug!
> Hello World Debug!
> Hello World Debug!
```

Now you can close the IDE and clean the temporary files:

```
# close VS IDE
$ git clean -xdf
```

Again, there is a **conanfile.py** package recipe in the repository, together with a **test_package**. The recipe is almost identical to the above one, just with two minor differences:
This will allow us to create and test the package of the ChatLib library:

```
$ conan create . memsharded/testing
> Hello World Release!
> Hello World Release!
```

You can also repeat the process for different build types and architectures.

### 13.3.4 Other configurations

The above example works as-is for VS2017, because VS supports upgrading from previous versions. The `MSBuild()` already implements such functionality, so building and testing packages with VS2017 can be done.

```
$ conan create . demo/testing -s compiler="Visual Studio" -s compiler.version=15
```

If you have to build for older versions of Visual Studio, it is also possible. In that case, you would probably have different solution projects inside your build folder. Then the recipe only has to select the correct one, something like:

```python
def build(self):
    # assuming HelloLibVS12, HelloLibVS14 subfolders
    sln_file = "build/HelloLibVS%s/HelloLib.sln" % self.settings.compiler.version
    msbuild = MSBuild(self)
    msbuild.build(sln_file)
```

Finally, we used just one `conanbuildinfo.props` file, which the solution loaded at a global level. You could also define multiple `conanbuildinfo.props` files, one per configuration (Release/Debug, x86/x86_64), and load them accordingly.

**Note:** So far, the `visual_studio` generator is single-configuration (packages containing debug or release artifacts, the generally recommended approach). It does not support multi-config packages (packages containing both debug and release artifacts). Please report and provide feedback (submit an issue in github) to request this feature if necessary.

### 13.4 Creating and reusing packages based on Makefiles

Conan can create packages and reuse them with Makefiles. The `AutoToolsBuildEnvironment` build helper helps with most of the necessary tasks.

This how-to has been tested in Windows with MinGW and Linux with gcc. It uses static libraries but could be extended to shared libraries too. The Makefiles surely can be improved. They are just an example.

#### 13.4.1 Creating packages

Start cloning the existing example repository, containing a simple “Hello World” library, and application:

```
$ git clone https://github.com/memsharded/conan-example-makefiles
$ cd conan-example-makefiles
$ cd hellolib
```
It contains a `src` folder with the source code and a `conanfile.py` file for creating a package.

Inside the `src` folder, there is a `Makefile` to build the static library. This `Makefile` uses standard variables like `$(CPPFLAGS)` or `$(CXX)` to build it:

```makefile
SRC = hello.cpp
OBJ = $(SRC:.cpp=.o)
OUT = libhello.a
INCLUDES = -I.

.SUFFIXES: .cpp

default: $(OUT)

.cpp.o:
  $(CXX) $(INCLUDES) $(CPPFLAGS) $(CXXFLAGS) -c $< -o $@

$(OUT):
  ar rcs $(OUT) $(OBJ)
```

The `conanfile.py` file uses the `AutoToolsBuildEnvironment` build helper. This helper defines the necessary environment variables with information from dependencies, as well as other variables to match the current Conan settings (like `-m32` or `-m64` based on the Conan `arch` setting):

```python
from conans import ConanFile, AutoToolsBuildEnvironment
from conans import tools

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

    def build(self):
        with tools.chdir("src"):
            env_build = AutoToolsBuildEnvironment(self)
            # env_build.configure() # use it to run "./configure" if using autotools
            env_build.make()

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

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

With this `conanfile.py` you can create the package:

```bash
$ conan create . user/testing -s compiler=gcc -s compiler.version=4.9 -s compiler.libcxx=libstdc++
```

### 13.4.2 Using packages

Now let’s move to the application folder:
$ cd ../helloapp

There you can also see a src folder with a Makefile creating an executable:

```
SRC = app.cpp
OBJ = $(SRC:.cpp=.o)
OUT = app
INCLUDES = -I.

.SUFFIXES: .cpp

default: $(OUT)

.cpp.o:
   $(CXX) $(CPPFLAGS) $(CXXFLAGS) -c $< -o $@

$(OUT): $(OBJ)
   $(CXX) -o $(OUT) $(OBJ) $(LDFLAGS) $(LIBS)
```

And also a `conanfile.py` very similar to the previous one. In this case adding a requires and a deploy() method:

```
from conans import ConanFile, AutoToolsBuildEnvironment
from conans import tools

class AppConan(ConanFile):
    name = "App"
    version = "0.1"
    settings = "os", "compiler", "build_type", "arch"
    exports_sources = "src/*"
    requires = "Hello/0.1@user/testing"

    def build(self):
        with tools.chdir("src"):
            env_build = AutoToolsBuildEnvironment(self)
            env_build.make()

    def package(self):
        self.copy("*app", dst="bin", keep_path=False)
        self.copy("*app.exe", dst="bin", keep_path=False)

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

Note that in this case, the AutoToolsBuildEnvironment will automatically set values to CPPFLAGS, LDFLAGS, LIBS, etc. existing in the Makefile with the correct include directories, library names, etc. to properly build and link with the hello library contained in the “Hello” package.

As above, we can create the package with:

```
$ conan create . user/testing -s compiler=gcc -s compiler.version=4.9 -s compiler.libcxx=libstdc++
```

There are different ways to run executables contained in packages, like using virtualrunenv generators. In this case, since the package has a deploy() method, we can use it:
13.5 How to manage the GCC >= 5 ABI

In version 5.1, GCC released libstdc++, which introduced a new library ABI that includes new implementations of `std::string` and `std::list`. These changes were necessary to conform to the 2011 C++ standard which forbids Copy-On-Write strings and requires lists to keep track of their size.

You can choose which ABI to use in your Conan packages by adjusting the `compiler.libcxx`:

- `libstdc++`: Old ABI.
- `libstdc++11`: New ABI.

When Conan creates the default profile the first time it runs, it adjusts the `compiler.libcxx` setting to `libstdc++` for backwards compatibility. However, if you are using GCC >= 5 your compiler is likely to be using the new CXX11 ABI by default (libstdc++11).

If you want Conan to use the new ABI, edit the default profile at `~/.conan/profiles/default` adjusting `compiler.libcxx=libstdc++11` or override this setting in the profile you are using.

If you are using the CMake build helper or the AutotoolsBuildEnvironment build helper Conan will automatically adjust the `_GLIBCXX_USE_CXX11_ABI` flag to manage the ABI.

13.6 Using Visual Studio 2017 - CMake integration

Visual Studio 2017 comes with a CMake integration that allows one to just open a folder that contains a `CMakeLists.txt` and Visual will use it to define the project build.

Conan can also be used in this setup to install dependencies. Let’s say that we are going to build an application that depends on an existing Conan package called `Hello/0.1@user/testing`. For the purpose of this example, you can quickly create this package by typing in your terminal:

```bash
$ conan new Hello/0.1 -s
$ conan create . user/testing # Default conan profile is Release
$ conan create . user/testing -s build_type=Debug
```

The project we want to develop will be a simple application with these 3 files in the same folder:

```cpp
// example.cpp
#include <iostream>
#include "hello.h"

int main() {
    hello();
    std::cin.ignore();
}
```
conan Documentation, Release 1.16.0

Listing 3: conanfile.txt

```conan
[requires]
Hello/0.1@user/testing

[generators]
cmake
```

Listing 4: CMakeLists.txt

```cmake
project(Example CXX)
cmake_minimum_required(VERSION 2.8.12)
include(${CMAKE_BINARY_DIR}/conanbuildinfo.cmake)
conan_basic_setup()

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

If we open Visual Studio 2017 (with CMake support installed), and select “Open Folder” from the menu, and select the above folder, we will see something like the following error:

```
1> Command line: C:\PROGRAM FILES (X86)\MICROSOFT VISUAL\STUDIO\2017\COMMUNITY\COMMON7\IDE\COMMONEXTENSIONS\MICROSOFT\CMAKE\CMake\bin\cmake.
exe -G "Ninja" -DCMAKE_INSTALL_PREFIX:PATH="C:\Users\user\CMakeBuilds\df6639d2-3ef2-bc32-abb3-2cd1bdb3c1ab\install\x64-Debug" -DCMAKE_CXX_COMPILER="C:/Program Files (x86)/Microsoft Visual Studio/2017/Community/VC/Tools/MSVC/14.12.25827/bin/HostX64/x64/cl.exe" -DCMAKE_C_COMPILER="C:/Program Files (x86)/Microsoft Visual Studio/2017/Community/VC/Tools/MSVC/14.12.25827/bin/HostX64/x64/cl.exe" -DCMAKE_BUILD_TYPE="Debug" -DCMAKE_MAKE_PROGRAM="C:\PROGRAM FILES (X86)\MICROSOFT VISUAL\STUDIO\2017\COMMUNITY\COMMON7\IDE\COMMONEXTENSIONS\MICROSOFT\CMAKE\Ninja\ninja.exe" -"C:\Users\user\cmakewws\visual-cmake"
1> Working directory: C:\Users\user\CMakeBuilds\df6639d2-3ef2-bc32-abb3-2cd1bdb3c1ab\build\x64-Debug
1> -- The CXX compiler identification is MSVC 19.12.25831.0
1> -- Check for working CXX compiler: C:\Program Files (x86)\Microsoft Visual Studio\2017\Community\VC\Tools\MSVC/14.12.25827\bin\HostX64\x64\cl.exe
1> -- Check for working CXX compiler: C:\Program Files (x86)\Microsoft Visual Studio\2017\Community\VC\Tools\MSVC/14.12.25827\bin\HostX64\x64\cl.exe -- works
1> -- Detecting CXX compiler ABI info
1> -- Detecting CXX compiler ABI info - done
1> -- Detecting CXX compiler ABI info - done
1> -- Detecting CXX compiler ABI info - done
1> CMake Error at CMakeLists.txt:4 (include):
1> include could not find load file:
1> C:\Users\user\CMakeBuilds\df6639d2-3ef2-bc32-abb3-2cd1bdb3c1ab\build\x64-Debug\conanbuildinfo.cmake
```

As expected, our `CMakeLists.txt` is using an `include(${CMAKE_BINARY_DIR}/conanbuildinfo.cmake)`, and that file doesn’t exist yet, because Conan has not yet installed the dependencies of this project. Visual Studio 2017 uses different build folders for each configuration. In this case, the default configuration at startup is x64-Debug. This means that we need to install the dependencies that match this configuration. Assuming that our default profile is using Visual Studio 2017 for x64 (it should typically be the default one created by Conan if VS2017 is present), then all we need to specify is the `-s build_type=Debug` setting:
conan Documentation, Release 1.16.0

$ conan install . -s build_type=Debug -if=C:\Users\user\CMakeBuilds\df6639d2-3ef2˓→bc32-abb3-2cd1bdb3c1ab\build\x64-Debug

Now, you should be able to regenerate the CMake project from the IDE, Menu->CMake, build it, select the “example”
executable to run, and run it.
Now, let’s say that you want to build the Release application. You switch configuration from the IDE, and then the
above error happens again. The dependencies for Release mode need to be installed too:
$ conan install . -if=C:\Users\user\CMakeBuilds\df6639d2-3ef2-bc32-abb3˓→2cd1bdb3c1ab\build\x64-Release

The process can be extended to x86 (passing -s arch=x86 in the command line), or to other configurations. For
production usage, Conan profiles are highly recommended.

13.6.1 Using cmake-conan
The cmake-conan project in https://github.com/conan-io/cmake-conan is a CMake script that runs an
execute_process that automatically launches conan install to install dependencies. The settings passed
in the command line will be derived from the current CMake configuration, that will match the Visual Studio one.
This script can be used to further automate the installation task:
project(Example CXX)
cmake_minimum_required(VERSION 2.8.12)
# Download automatically, you can also just copy the conan.cmake file
if(NOT EXISTS "${CMAKE_BINARY_DIR}/conan.cmake")
message(STATUS "Downloading conan.cmake from https://github.com/conan-io/cmake-conan")
file(DOWNLOAD "https://raw.githubusercontent.com/conan-io/cmake-conan/v0.9/conan.
˓→cmake"
"${CMAKE_BINARY_DIR}/conan.cmake")
endif()
include(${CMAKE_BINARY_DIR}/conan.cmake)
conan_cmake_run(CONANFILE conanfile.txt
BASIC_SETUP)
add_executable(example example.cpp)
target_link_libraries(example ${CONAN_LIBS})

This code will manage to download the cmake-conan CMake script, and use it automatically, calling a conan
install automatically.
There could be an issue, though, for the Release configuration. Internally, the Visual Studio 2017 defines the
configurationType As RelWithDebInfo for Release builds. But Conan default settings (in the Conan
settings.yml file), only have Debug and Release defined. It is possible to modify the settings.yml file, and add
those extra build types. Then you should create the Hello package for those settings. And most existing packages,
specially in central repositories, are built only for Debug and Release modes.
An easier approach is to change the CMake configuration in Visual: go to the Menu -> CMake -> Change CMake
Configuration. That should open the CMakeSettings.json file, and there you can change the configurationType
to Release:
{
"name": "x64-Release",
(continues on next page)

226

Chapter 13. Howtos


"generator": "Ninja",
"configurationType": "Release",
"inheritEnvironments": [ "msvc_x64_x64" ],
"buildRoot": "${env.USERPROFILE}\CMakeBuilds\${workspaceHash}\build\${name}",
"installRoot": "${env.USERPROFILE}\CMakeBuilds\${workspaceHash}\install\${name}"
}

Note that the above CMake code is only valid for consuming existing packages. If you are also creating a package, you would need to make sure the right CMake code is executed, please check https://github.com/conan-io/cmake-conan/blob/master/README.md

13.6.2 Using tasks with tasks.vs.json

Another alternative is using file tasks feature of Visual Studio 2017. This way you can install dependencies by running conan install as task directly in the IDE.

All you need is to right click on your conanfile.py-> Configure Tasks (see the link above) and add the following to your tasks.vs.json.

Warning: The file tasks.vs.json is added to your local .vs folder so it is not supposed to be added to your version control system. There is also a feature request to improve this process.

```json
{
   "tasks": [
      {
         "taskName": "conan install debug",
         "appliesTo": "conanfile.py",
         "type": "launch",
         "command": "${env.COMSPEC}",
         "args": [
            "conan install ${file} -s build_type=Debug -if C:/Users/user/CMakeBuilds/4c2d87b9-ec5a-9a30-a47a-32cbb6cca172/build/x64-Debug/"
         ]
      },
      {
         "taskName": "conan install release",
         "appliesTo": "conanfile.py",
         "type": "launch",
         "command": "${env.COMSPEC}",
         "args": [
            "conan install ${file} -s build_type=Release -if C:/Users/user/CMakeBuilds/4c2d87b9-ec5a-9a30-a47a-32cbb6cca172/build/x64-Release/"
         ]
      }
   ],
   "version": "0.2.1"
}
```

Then just right click on your conanfile.py and launch your conan install and regenerate your CMakeLists.txt.
13.7 How to manage C++ standard [EXPERIMENTAL]

Warning: This is an experimental feature subject to breaking changes in future releases. Previously, it was implemented as a first level setting `cppstd`, we encourage you to adopt the new subsetting and update your recipes if they were including the deprecated one in its `settings` attribute.

The setting representing the C++ standard is `compiler.cppstd`. The detected default profile doesn’t set any value for the `compiler.cppstd` setting.

The consumer can specify it in a profile or with the `-s` parameter:

```conan install . -s compiler.cppstd=gnu14```

As it is a subsetting, it can have different values for each compiler (also, take into account that depending on the version of the compiler the standard could have only partial support and may change the ABI).

Valid values for `compiler=Visual Studio`:

<table>
<thead>
<tr>
<th>VALUE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>C++ 14</td>
</tr>
<tr>
<td>17</td>
<td>C++ 17</td>
</tr>
<tr>
<td>20</td>
<td>C++ 20 (Still C++20 Working Draft)</td>
</tr>
</tbody>
</table>

Valid values for other compilers:

<table>
<thead>
<tr>
<th>VALUE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>98</td>
<td>C++ 98</td>
</tr>
<tr>
<td>gnu98</td>
<td>C++ 98 with GNU extensions</td>
</tr>
<tr>
<td>11</td>
<td>C++ 11</td>
</tr>
<tr>
<td>gnu11</td>
<td>C++ 11 with GNU extensions</td>
</tr>
<tr>
<td>14</td>
<td>C++ 14</td>
</tr>
<tr>
<td>gnu14</td>
<td>C++ 14 with GNU extensions</td>
</tr>
<tr>
<td>17</td>
<td>C++ 17</td>
</tr>
<tr>
<td>gnu17</td>
<td>C++ 17 with GNU extensions</td>
</tr>
<tr>
<td>20</td>
<td>C++ 20 (Partial support)</td>
</tr>
<tr>
<td>gnu20</td>
<td>C++ 20 with GNU extensions (Partial support)</td>
</tr>
</tbody>
</table>

13.7.1 Build helpers

The value of `compiler.cppstd` provided by the consumer is used by the build helpers:

- The `CMake` build helper will set the `CONAN_CMAKE_CXX_STANDARD` and `CONAN_CMAKE_CXX_EXTENSIONS` definitions that will be converted to the corresponding CMake variables to activate the standard automatically with the `conan_basic_setup()` macro.
- The `AutotoolsBuildEnvironment` build helper will adjust the needed flag to `CXXFLAGS` automatically.
- The `MSBuild/VisualStudioBuildEnvironment` build helper will adjust the needed flag to `CL` env var automatically.
13.7.2 Package compatibility

By default Conan will detect the default standard of your compiler to not generate different binary packages. For example, you already built some gcc 6.1 packages, where the default C++ standard is gnu14. If you introduce the compiler.cppstd setting in your profile with the gnu14 value, Conan won’t generate new packages, because it was already the default of your compiler.

**Note:** Check the `package_id()` reference to know more.

**Note:** Conan 1.x will also generate the same packages as the ones generated with the deprecated setting `cppstd` for the default value of the standard.

13.8 How to use Docker to create and cross-build C and C++ Conan packages

With Docker, you can run different virtual Linux operating systems in a Linux, Mac OSX or Windows machine. It is useful to reproduce build environments, for example to automate CI processes. You can have different images with different compilers or toolchains and run containers every time is needed.

In this section you will find a list of pre-built images with common build tools and compilers as well as Conan installed.

13.8.1 Using Conan inside a container

```
$ docker run -it --rm conanio/gcc7 /bin/bash
```

**Note:** Use `sudo` when needed to run `docker`.

The previous code will run a shell in container. We have specified:

- **-it**: Keep STDIN open and allocate a pseudo-tty, in other words, we want to type in the container because we are opening a bash.
- **--rm**: Once the container exits, remove the container. Helps to keep clean or hard drive.
- **conanio/gcc7**: Image name, check the available Docker images.
- **/bin/bash**: The command to run

Now we are running on the conangcc7 container we can use Conan normally. In the following example we are creating a package from the recipe by cloning the repository, for OpenSSL. It is always recommended to upgrade Conan from pip first:

```
$ sudo pip install conan --upgrade # We make sure we are running the latest Conan
$ git clone https://github.com/conan-community/conan-openssl
$ cd conan-openssl
$ conan create . user/channel
```
13.8.2 Sharing a local folder with a Docker container

You can share a local folder with your container, for example a project:

```bash
$ git clone https://github.com/conan-community/conan-openssl
$ cd conan-openssl
$ docker run -it -v $(pwd)/:home/conan/project --rm conanio/gcc7 /bin/bash
```

- `-v $(pwd)/:home/conan/project`: We are mapping the current directory (conan-openssl) to the container /home/conan/project directory, so anything we change in this shared folder, will be reflected in our host machine.

```bash
# Now we are running on the conangcc7 container
$ sudo pip install conan --upgrade # We make sure we are running the latest Conan
$ cd project
$ conan create . user/channel --build missing
$ conan remote add myremote http://some.remote.url
$ conan upload "*" -r myremote --all
```

13.8.3 Using the images to cross-build packages

You can use the `images` `-i386`, `-armv7` and `-armv7gh` to cross-build Conan packages.

The `armv7` images have a cross toolchain for linux ARM installed, and declared as main compiler with the environment variables `CC` and `CXX`. Also, the default Conan profile (`~/.conan/profiles/default`) is adjusted to declare the correct arch (`armv7`/`armv7hf`).

Cross-building and uploading a package along with all its missing dependencies for Linux/armv7hf is done in few steps:

```bash
$ git clone https://github.com/conan-community/conan-openssl
$ cd conan-openssl
$ docker run -it -v $(pwd)/:home/conan/project --rm conanio/gcc49-armv7hf /bin/bash
# Now we are running on the conangcc49-armv7hf container
# The default profile is automatically adjusted to armv7hf
$ cat ~/.conan/profiles/default
[settings]
os=Linux
os_build=Linux
arch=armv7hf
arch_build=x86_64
compiler=gcc
compiler.version=4.9
compiler.libcxx=libstdc++
build_type=Release
[options]
[buildRequires]
[env]
$ sudo pip install conan --upgrade # We make sure we are running the latest Conan
$ cd project
```

(continues on next page)
13.8.4 Available Docker images

GCC images

<table>
<thead>
<tr>
<th>Version</th>
<th>Target Arch</th>
</tr>
</thead>
<tbody>
<tr>
<td>conanio/gcc49 (GCC 4.9)</td>
<td>x86_64</td>
</tr>
<tr>
<td>conanio/gcc49-i386 (GCC 4.9)</td>
<td>x86</td>
</tr>
<tr>
<td>conanio/gcc49-armv7 (GCC 4.9)</td>
<td>armv7</td>
</tr>
<tr>
<td>conanio/gcc49-armv7hf (GCC 4.9)</td>
<td>armv7hf</td>
</tr>
<tr>
<td>conanio/gcc5-armv7 (GCC 5)</td>
<td>armv7</td>
</tr>
<tr>
<td>conanio/gcc5-armv7hf (GCC 5)</td>
<td>armv7hf</td>
</tr>
<tr>
<td>conanio/gcc5-i386 (GCC 5)</td>
<td>x86</td>
</tr>
<tr>
<td>conanio/gcc5-armv7 (GCC 5)</td>
<td>armv7</td>
</tr>
<tr>
<td>conanio/gcc5-armv7hf (GCC 5)</td>
<td>armv7hf</td>
</tr>
<tr>
<td>conanio/gcc6 (GCC 6)</td>
<td>x86_64</td>
</tr>
<tr>
<td>conanio/gcc6-i386 (GCC 6)</td>
<td>x86</td>
</tr>
<tr>
<td>conanio/gcc6-armv7 (GCC 6)</td>
<td>armv7</td>
</tr>
<tr>
<td>conanio/gcc6-armv7hf (GCC 6)</td>
<td>armv7hf</td>
</tr>
<tr>
<td>conanio/gcc7-i386 (GCC 7)</td>
<td>x86</td>
</tr>
<tr>
<td>conanio/gcc7 (GCC 7)</td>
<td>x86_64</td>
</tr>
<tr>
<td>conanio/gcc7-armv7 (GCC 7)</td>
<td>armv7</td>
</tr>
<tr>
<td>conanio/gcc7-armv7hf (GCC 7)</td>
<td>armv7hf</td>
</tr>
</tbody>
</table>

Clang images

<table>
<thead>
<tr>
<th>Version</th>
<th>Target Arch</th>
</tr>
</thead>
<tbody>
<tr>
<td>conanio/clang38 (Clang 3.8)</td>
<td>x86_64</td>
</tr>
<tr>
<td>conanio/clang39-i386 (Clang 3.9)</td>
<td>x86</td>
</tr>
<tr>
<td>conanio/clang39 (Clang 3.9)</td>
<td>x86_64</td>
</tr>
<tr>
<td>conanio/clang40-i386 (Clang 4)</td>
<td>x86</td>
</tr>
<tr>
<td>conanio/clang40 (Clang 4)</td>
<td>x86_64</td>
</tr>
<tr>
<td>conanio/clang50-i386 (Clang 5)</td>
<td>x86</td>
</tr>
<tr>
<td>conanio/clang50 (Clang 5)</td>
<td>x86_64</td>
</tr>
</tbody>
</table>

The Dockerfiles for all these images can be found here.

13.9 How to reuse Python code in recipes

**Warning:** To reuse Python code, from Conan 1.7 there is a new `python_requires()` feature. See: [Python requires: reusing Python code in recipes](#). This “how to” might be deprecated and removed in the future. It is left here for reference only.
First, if you feel that you are repeating a lot of Python code, and that repeated code could be useful for other Conan users, please propose it in a github issue.

There are several ways to handle Python code reuse in package recipes:

- To put common code in files, as explained below. This code has to be exported into the recipe itself.
- To create a Conan package with the common Python code, and then require it from the recipe.

This howto explains the latter.

### 13.9.1 A basic Python package

Let’s begin with a simple Python package, a “hello world” functionality that we want to package and reuse:

```python
def hello():
    print("Hello World from Python!")
```

To create a package, all we need to do is create the following layout:

```
-| hello.py
 | __init__.py
 | conanfile.py
```

The `__init__.py` is blank. It is not necessary to compile code, so the package recipe `conanfile.py` is quite simple:

```python
from conans import ConanFile

class HelloPythonConan(ConanFile):
    name = "HelloPy"
    version = "0.1"
    exports = '*'
    build_policy = "missing"

    def package(self):
        self.copy('*.py')

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

The `exports` will copy both the `hello.py` and the `__init__.py` into the recipe. The `package()` method is also obvious: to construct the package just copy the Python sources.

The `package_info()` adds the current package folder to the `PYTHONPATH` Conan environment variable. It will not affect the real environment variable unless the end user desires it.

It can be seen that this recipe would be practically the same for most Python packages, so it could be factored in a `PythonConanFile` base class to further simplify it. (Open a feature request, or better a pull request. :)

With this recipe, all we have to do is:

```
$ conan export . memsharded/testing
```

Of course if you want to share the package with your team, you can `conan upload` it to a remote server. But to create and test the package, we can do everything locally.

Now the package is ready for consumption. In another folder, we can create a `conanfile.txt` (or a `conanfile.py` if we prefer):
And install it with the following command:

```bash
$ conan install . -g virtualenv
```

Creating the above `conanfile.txt` might be unnecessary for this simple example, as you can directly run `conan install HelloPy/0.1@memsharded/testing -g virtualenv`, however, using the file is the canonical way.

The specified `virtualenv` generator will create an `activate` script (in Windows `activate.bat`), that basically contains the environment, in this case, the `PYTHONPATH`. Once we activate it, we are able to find the package in the path and use it:

```bash
$ activate
$ python
Python 2.7.12 (v2.7.12:d33e0cf91556, Jun 27 2016, 15:19:22) [MSC v.1500 32 bit (Intel)] on win32
...
>>> import hello
>>> hello.hello()
Hello World from Python!
```

The above shows an interactive session, but you can import also the functionality in a regular Python script.

## 13.9.2 Reusing Python code in your recipes

### Requiring a Python Conan package

As the Conan recipes are Python code itself, it is easy to reuse Python packages in them. A basic recipe using the created package would be:

```python
from conans import ConanFile

class HelloPythonReuseConan(ConanFile):
    requires = "HelloPy/0.1@memsharded/testing"

    def build(self):
        from hello import hello
        hello()
```

The `requires` section is just referencing the previously created package. The functionality of that package can be used in several methods of the recipe: `source()`, `build()`, `package()` and `package_info()`, i.e. all of the methods used for creating the package itself. Note that in other places it is not possible, as it would require the dependencies of the recipe to be already retrieved, and such dependencies cannot be retrieved until the basic evaluation of the recipe has been executed.

```bash
$ conan install .
...
$ conan build .
Hello World from Python!
```
Sharing a Python module

Another approach is sharing a Python module and exporting within the recipe.

Let’s write for example a `msgs.py` file and put it besides the `conanfile.py`:

```python
def build_msg(output):
    output.info("Building!")
```

And then the main `conanfile.py` would be:

```python
from conans import ConanFile
from msgs import build_msg

class ConanFileToolsTest(ConanFile):
    name = "test"
    version = "1.9"
    exports = "msgs.py"  # Important to remember!

    def build(self):
        build_msg(self.output)
        # ...
```

It is important to note that such `msgs.py` file must be exported too when exporting the package, because package recipes must be self-contained.

The code reuse can also be done in the form of a base class, something like a file `base_conan.py`

```python
from conans import ConanFile

class ConanBase(ConanFile):
    # common code here
```

And then:

```python
from conans import ConanFile
from base_conan import ConanBase

class ConanFileToolsTest(ConanBase):
    name = "test"
    version = "1.9"
    exports = "base_conan.py"
```

13.10 How to create and share a custom generator with generator packages

There are several built-in generators, like `cmake`, `visual_studio`, `xcode`... But what if your build system is not included or the existing built-in ones doesn’t satisfy your needs? This how to will show you how to create a generator for Premake build system.

Important: Check the reference of the custom_generator section to know the syntax and attributes available.
13.10.1 Creating a Premake generator

Create a folder with a new `conanfile.py` with the following contents:

```bash
$ mkdir conan-premake && cd conan-premake
```

Listing 5: `conanfile.py`

```python
from conans.model import Generator
from conans import ConanFile

class PremakeDeps(object):
    def __init__(self, deps_cpp_info):
        self.include_paths = '
            '.join('"%s"' % p.replace(r'\', '/')
                for p in deps_cpp_info.include_paths)
        self.lib_paths = '
            '.join('"%s"' % p.replace(r'\', '/')
                for p in deps_cpp_info.lib_paths)
        self.bin_paths = '
            '.join('"%s"' % p.replace(r'\', '/')
                for p in deps_cpp_info.bin_paths)
        self.libs = ', '.join('"%s"' % p
            for p in deps_cpp_info.libs)
        self.defines = ', '.join('"%s"' % p
            for p in deps_cpp_info.defines)
        self.cppflags = ', '.join('"%s"' % p
            for p in deps_cpp_info.cppflags)
        self.cflags = ', '.join('"%s"' % p
            for p in deps_cpp_info.cflags)
        self.sharedlinkflags = ', '.join('"%s"' % p
            for p in deps_cpp_info.sharedlinkflags)
        self.exelinkflags = ', '.join('"%s"' % p
            for p in deps_cpp_info.exelinkflags)
        self.rootpath = "%s" % deps_cpp_info.rootpath.replace(r'\', '/

class Premake(Generator):

    @property
    def filename(self):
        return "conanpremake.lua"

    @property
    def content(self):
        deps = PremakeDeps(self.deps_build_info)

        template = ('"conan_includedirs{dep} = {{{deps.include_paths}}}"
'"conan_libdirs{dep} = {{{deps.lib_paths}}}"
'"conan_bindedirs{dep} = {{{deps.bin_paths}}}"
'"conan_libs{dep} = {{{deps.libs}}}"
'"conan_cppdefines{dep} = {{{deps.defines}}}"
'"conan_cppflags{dep} = {{{deps.cppflags}}}"
'"conan_cflags{dep} = {{{deps.cflags}}}"
'"conan_sharedlinkflags{dep} = {{{deps.sharedlinkflags}}}"
'"conan_exelinkflags{dep} = {{{deps.exelinkflags}}}"

        sections = ["#!lua"]
        all_flags = template.format(dep="", deps=deps)
        sections.append(all_flags)
        template_deps = template + "conan_rootpath{dep} = ""(deps.rootpath)"

        for dep_name, dep_cpp_info in self.deps_build_info.dependencies:
            sections.append(all_flags)
            template_deps += template + 'conan_rootpath{dep} = "(deps.rootpath)"
            ...
```

(continues on next page)
This is a full working example. Note the `PremakeDeps` class as a helper. The generator is creating Premake information for each individual library separately, then also an aggregated information for all dependencies. This `PremakeDeps` wraps a single item of such information.

Note the **name of the package** will be `PremakeGen/0.1@<user>/<channel>` as that is the name given to it, while the generator name is `Premake` (the name of the class that inherits from `Generator`). You can give the package any name you want, even the same as the generator’s name if desired.

You export the package recipe to the local cache, so it can be used by other projects as usual:

```bash
$ conan export . myuser/testing
```

### 13.10.2 Using the generator

Let’s create a test project that uses this generator. We will use a simple application that will use a “Hello World” library package as a requirement.

First, let’s create the “Hello World” library package:

```bash
$ mkdir conan-hello && cd conan-hello
$ conan new hello/0.1
$ conan create . myuser/testing
```

Now, let’s create a folder for the application that will use Premake as build system:

```bash
$ cd ..
$ mkdir premake-project && cd premake-project
```

Put the following files inside. Note the `PremakeGen@0.1@myuser/testing` package reference in your `conanfile.txt`.

**Listing 6: conanfile.txt**

```yaml
[requires]
hello/0.1@myuser/testing
PremakeGen@0.1@myuser/testing

[generators]
Premake
```
### Listing 7: main.cpp

```cpp
#include "hello.h"

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

### Listing 8: premake4.lua

```lua
-- premake4.lua

require 'conanpremake'

-- A solution contains projects, and defines the available configurations solution
→ "MyApplication"

configurations { "Debug", "Release" }
includedirs { conan_includedirs }
libdirs { conan_libdirs }
links { conan_libs }

-- A project defines one build target

project "MyApplication"
    kind "ConsoleApp"
    language "C++"
    files { "**.h", "**.cpp" }

    configuration "Debug"
        defines { "DEBUG" }
        flags { "Symbols" }

    configuration "Release"
        defines { "NDEBUG" }
        flags { "Optimize" }
```

Let’s install the requirements:

```sh
$ conan install . -s compiler=gcc -s compiler.version=4.9 -s compiler.libcxx=libstdc++ --build
```

This generates the `premake4.lua` file with the requirements information for building.

Now we are ready to build the project:

```sh
$ premake4 gmake
$ make (or mingw32-make `if in windows-mingw)
$ ./MyApplication
Hello World!
```

Now everything works, so you might want to share your generator:

```sh
$ conan upload PremakeGen/0.1@myuser/testing
```

**Tip:** This is a regular Conan package, so you could create a `test_package` folder with a `conanfile.py` to test the
13.10.3 Using template files for custom generators

If your generator has a lot of common, non-parameterized text, you might want to use files that contain the template. It is possible to do this as long as the template file is exported in the recipe. The following example uses a simple text file, but you could use other templating formats:

```python
import os
from conans import ConanFile, load
from conans.model import Generator

class MyCustomGenerator(Generator):
    @property
def filename(self):
        return "customfile.gen"

    @property
def content(self):
        template = load(os.path.join(os.path.dirname(__file__), "mytemplate.txt"))
        return template % "Hello"

class MyCustomGeneratorPackage(ConanFile):
    name = "custom"
    version = "0.1"
    exports = "mytemplate.txt"
```

13.11 How to manage shared libraries

Shared libraries, `.dll` in windows, `.dylib` in OSX and `.so` in Linux, are loaded at runtime. That means that the application executable needs to know where are the required shared libraries when it runs.

On Windows, the dynamic linker, will search in the same directory then in the `PATH` directories. On OSX, it will search in the directories declared in `DYLD_LIBRARY_PATH` as on Linux will use the `LD_LIBRARY_PATH`.

Furthermore in OSX and Linux there is another mechanism to locate the shared libraries: The RPATHs.

13.11.1 Manage Shared Libraries with Environment Variables

Shared libraries are loaded at runtime. The application executable needs to know where to find the required shared libraries when it runs.

Depending on the operating system, we can use environment variables to help the dynamic linker to find the shared libraries:

<table>
<thead>
<tr>
<th>OPERATING SYSTEM</th>
<th>ENVIRONMENT VARIABLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>WINDOWS</td>
<td>PATH</td>
</tr>
<tr>
<td>LINUX</td>
<td>LD_LIBRARY_PATH</td>
</tr>
<tr>
<td>OSX</td>
<td>DYLD_LIBRARY_PATH</td>
</tr>
</tbody>
</table>
If your package recipe (A) is generating shared libraries you can declare the needed environment variables pointing to the package directory. This way, any other package depending on (A) will automatically have the right environment variable set, so they will be able to locate the (A) shared library.

Similarly if you use the virtualenv generator and you activate it, you will get the paths needed to locate the shared libraries in your terminal.

**Example**

We are packaging a tool called `toolA` with a library and an executable that will, for example, compress data.

The package offers two flavors, shared library or static library (embedded in the executable of the tool and available to link with). You can use the `toolA` package library to develop another executable or library or you can just use the executable provided by the package. In both cases, if you choose to install the shared package of `toolA` you will need to have the shared library available.

```python
import os
from conans import tools, ConanFile

class ToolA(ConanFile):
    ....
    name = "toolA"
    version = "1.0"
    options = {"shared": [True, False]}
    default_options = {"shared": False}

    def build(self):
        # build your shared library

    def package(self):
        # Copy the executable
        self.copy(pattern="toolA*", dst="bin", keep_path=False)

        # Copy the libraries
        if self.options.shared:
            self.copy(pattern="*.dll", dst="bin", keep_path=False)
            self.copy(pattern="*.dylib", dst="lib", keep_path=False)
            self.copy(pattern="*.so*", dst="lib", keep_path=False)
        else:
            ...
```

**Using the tool from a different package**

If we are now creating a package that uses the `ToolA` executable to compress some data, we can execute directly `toolA` using RunEnvironment build helper to set the environment variables accordingly:

```python
import os
from conans import tools, ConanFile

class PackageB(ConanFile):
    name = "packageB"
    version = "1.0"
    requires = "toolA/1.0@myuser/stable"

    def build(self):
```

(continues on next page)
Building an application using the shared library from toolA

As we are building a final application, we will probably want to distribute it together with the shared library from the toolA, so we can use the **Imports** to import the required shared libraries to our user space.

Listing 9: conanfile.txt

```ini
[requires]
toolA/1.0@myuser/stable

[generators]
cmake

[options]
toolA:shared=True

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

Now you can build the project:

```bash
$ mkdir build && cd build
$ conan install ..
$ cmake .. -G "Visual Studio 14 Win64"
$ cmake --build . --config Release
$ cd bin && mytool
```

The previous example will work only in Windows and OSX (changing the CMake generator), because the dynamic linker will look in the current directory (the binary directory) where we copied the shared libraries too.

In Linux you still need to set the **LD_LIBRARY_PATH**, or in OSX, the **DYLD_LIBRARY_PATH**:

```bash
$ cd bin && LD_LIBRARY_PATH=$(pwd) && ./mytool
```

Using shared libraries from dependencies

If you are executing something that depends on shared libraries belonging to your dependencies, those shared libraries have to be found at runtime. In Windows, it is enough if the package added its binary folder to the system **PATH**. In Linux and OSX, it is necessary that the **LD_LIBRARY_PATH** and **DYLD_LIBRARY_PATH** environment variables are used.

Security restrictions might apply in OSX (read this thread), so the **DYLD_LIBRARY_PATH** environment variable is not directly transferred to the child process. In that case, you have to use it explicitly in your **conanfile.py**:
```python
def test(self):
    # self.run('./myexe') # won't work, even if 'DYLD_LIBRARY_PATH' is in the env
    with tools.environment_append({'DYLD_LIBRARY_PATH': [self.deps_cpp_info['toolA'].lib_paths]}):
        self.run('DYLD_LIBRARY_PATH=%s ./myexe' % os.environ['DYLD_LIBRARY_PATH'])
```

Or you could use RunEnvironment helper described above.

**Using virtualrunenv generator**

*virtualrunenv generator* will set the environment variables PATH, LD_LIBRARY_PATH, DYLD_LIBRARY_PATH pointing to *lib* and *bin* folders automatically.

Listing 10: *conanfile.txt*

```plaintext
[requires]
toolA/1.0@myuser/stable

[options]
toolA:shared=True

[generators]
virtualrunenv
```

In the terminal window:

```
$ conan install .
$ source activate_run
$ toolA --someparams
# Only For Mac OS users to avoid restrictions:
$ DYLD_LIBRARY_PATH=$DYLD_LIBRARY_PATH toolA --someparams
```

### 13.11.2 Manage RPATHs

The *rpath* is encoded inside dynamic libraries and executables and helps the linker to find its required shared libraries.

If we have an executable, *my_exe*, that requires a shared library, *shared_lib_1*, and *shared_lib_1*, in turn, requires another *shared_lib_2*.

So the *rpaths* values are:

<table>
<thead>
<tr>
<th>File</th>
<th>rpath</th>
</tr>
</thead>
<tbody>
<tr>
<td>my_exe</td>
<td>/path/to/shared_lib_1</td>
</tr>
<tr>
<td>shared_lib_1</td>
<td>/path/to/shared_lib_2</td>
</tr>
<tr>
<td>shared_lib_2</td>
<td></td>
</tr>
</tbody>
</table>

In *Linux* if the linker doesn’t find the library in *rpath*, it will continue the search in *system defaults paths* (LD_LIBRARY_PATH... etc) In *OSX*, if the linker detects an invalid *rpath* (the file does not exist there), it will fail.
**Default Conan approach**

The consumer project of dependencies with shared libraries needs to import them to the executable directory to be able to run it:

conanfile.txt

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

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

On Windows this approach works well, importing the shared library to the directory containing your executable is a very common procedure.

On Linux there is an additional problem, the dynamic linker doesn’t look by default in the executable directory, and you will need to adjust the `LD_LIBRARY_PATH` environment variable like this:

```
LD_LIBRARY_PATH=$(/bin) && ./mybin
```

On OSX if absolute rpaths are hardcoded in an executable or shared library and they don’t exist the executable will fail to run. This is the most common problem when we reuse packages in a different environment from where the artifacts have been generated.

So for OSX, Conan, by default, when you build your library with CMake, the rpaths will be generated without any path:

```
File      rpath
my_exe    shared_lib_1.dylib
shared_lib_1.dylib shared_lib_2.dylib
shared_lib_2.dylib
```

The `conan_basic_setup()` macro will set the `set(CMAKE_SKIP_RPATH 1)` in OSX.

You can skip this default behavior by passing the `KEEP_RPATHS` parameter to the `conan_basic_setup` macro:

```
include(${CMAKE_BINARY_DIR}/conanbuildinfo.cmake)
conan_basic_setup(KEEP_RPATHS)
add_executable(timer timer.cpp)
target_link_libraries(timer ${CONAN_LIBS})
```

If you are using autotools Conan won’t auto-adjust the rpaths behavior. if you want to follow this default behavior you will probably need to replace the `install_name` in the `configure` or `MakeFile` generated files in your recipe to not use $rpath:

```
replace_in_file("./configure", ":$rpath/", ":/"")
```

**Different approaches**

You can adjust the `rpaths` in the way that adapts better to your needs.

If you are using CMake take a look to the CMake RPATH handling guide.
Remember to pass the `KEEP_RPATHS` variable to the `conan_basic_setup`:

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

Then, you could, for example, use the `@executable_path` in OSX and `$ORIGIN` in Linux to adjust a relative path from the executable. Also, enabling `CMAKE_BUILD_WITH_INSTALL_RPATH` will build the application with the RPATH value of `CMAKE_INSTALL_RPATH` and avoid the need to be relinked when installed.

```cmake
if (APPLE)
    set(CMAKE_INSTALL_RPATH "@executable_path/../lib")
else()
    set(CMAKE_INSTALL_RPATH "$ORIGIN/../lib")
endif()
set(CMAKE_BUILD_WITH_INSTALL_RPATH ON)
```

You can use this imports statements in the consumer project:

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

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

And your final application can follow this layout:

```bash
bin
  |_____ my_executable
  |_____ mylib.dll
|
lib
  |_____ libmylib.so
  |_____ libmylib.dylib
```

You could move the entire application folder to any location and the shared libraries will be located correctly.

## 13.12 How to reuse cmake install for package() method

It is possible that your project’s `CMakeLists.txt` has already defined some functionality that extracts the artifacts (headers, libraries, binaries) from the build and source folder to a predetermined place and does the post-processing (e.g., strips rpaths). For example, one common practice is to use CMake `install` directive to that end.

When using Conan, the install phase of CMake is wrapped in the `package()` method. That way the flags like `conan create --keep-build` or the commands for the `Package development flow` are consistent with every step of the packaging process.

The following excerpt shows how to build and package with CMake within Conan. Mind that you need to configure CMake both in `build()` and in `package()`, since these methods are called independently.

```python
def _configure_cmake(self):
    cmake = CMake(self)
    cmake.definitions["SOME_DEFINITION"] = True
```

(continues on next page)
The `package_info()` method specifies the list of the necessary libraries, defines and flags for different build configurations for the consumers of the package. This is necessary as there is no possible way to extract this information from the CMake install automatically.

# 13.13 How to collaborate with other users’ packages

If a certain existing package does not work for you, or you need to store pre-compiled binaries for a platform not provided by the original package creator, you might still be able to do so:

## 13.13.1 Collaborate from source repository

If the original package creator has the package recipe in a repository, this would be the simplest approach. Just clone the package recipe on your machine, change something if you want, and then export the package recipe under your own user name. Point your project’s `[requires]` to the new package name, and use it as usual:

```bash
$ git clone <repository>
$ cd <repository>
//make changes if desired
$ conan export . <youruser/yourchannel>
```

You can just directly run:

```bash
$ conan create . demo/testing
```

Once you have generated the desired binaries, you can store your pre-compiled binaries in your bintray repository or on your own Conan server:

```bash
$ conan upload Package/0.1@myuser/stable -r=myremote --all
```

Finally, if you made useful changes, you might want to create a pull request to the original repository of the package creator.

## 13.13.2 Copy a package

If you don’t need to modify the original package creator recipe, it is fine to just copy the package to your local storage. You can copy the recipes and existing binary packages. This could be enough for caching existing binary packages from the original remote into your own remote, under your own username:

```bash
$ git clone <repository>
$ cd <repository>
//make changes if desired
$ conan export . <youruser/yourchannel>
```

You can just directly run:

```bash
$ conan create . demo/testing
```

Once you have generated the desired binaries, you can store your pre-compiled binaries in your bintray repository or on your own Conan server:

```bash
$ conan upload Package/0.1@myuser/stable -r=myremote --all
```
13.14 How to link with Apple Frameworks

It is common in MacOS that your Conan package needs to link with a complete Apple framework, and, of course, you want to propagate this information to all projects/libraries that use your package.

With regular libraries, use `self.cpp_info.libs` object to append to it all the libraries:

```python
def package_info(self):
    self.cpp_info.libs = ["SDL2"]
    self.cpp_info.libs.append("OpenGL32")
```

With frameworks we need to declare the “-framework flag” as a linker flag:

```python
def package_info(self):
    self.cpp_info.libs = ["SDL2"]
    self.cpp_info.exelinkflags.append("-framework Carbon")
    self.cpp_info.exelinkflags.append("-framework CoreAudio")
    self.cpp_info.exelinkflags.append("-framework Security")
    self.cpp_info.exelinkflags.append("-framework IOKit")
    self.cpp_info.sharedlinkflags = self.cpp_info.exelinkflags
```

In the previous example we are using `self.cpp_info.exelinkflags`. If we are using CMake to consume this package, it will only link those frameworks if we are building an executable and `sharedlinkflags` will only apply if we are building a shared library.

If we are not using CMake to consume this package `sharedlinkflags` and `exelinkflags` are used indistinctly. In the example above, in the last line we are assigning `sharedlinkflags` with `exelinkflags`, so no matter what the consumer will build, it will indicate to the linker to link with the specified frameworks.

13.15 How to package Apple Frameworks

To package a MyFramework Apple framework, copy/create a folder MyFramework.framework to your package folder, where you should put all the subdirectories (Headers, Modules, etc).

```python
def package(self):
    # If you have the framework folder built in your build_folder:
    self.copy("MyFramework.framework/*", symlinks=True)
    # Or build the destination folder:
    tools.mkdir("MyFramework.framework/Headers")
    self.copy("*.h", dst="MyFramework.framework/Headers")
    # ...
```

Declare the framework in the `cpp_info` object, pass a compiler flag `-F` with the directory of the framework folder (self.package_folder) and linker flags with the `-F` and `-framework` with the framework name.
def package_info(self):
    ...
    # Note that -F flags are not automatically adjusted in "cmake"
    # generator so it will be needed to declare its path like this:
    # self.cpp_info.exelinkflags.append("-F path/to/the/framework -framework
    # MyFramework")
    f_location = '-F "%s" ' % self.package_folder
    self.cpp_info.cflags.append(f_location) # or cpp_info.cppflags if cpp library
    self.cpp_info.sharedlinkflags.extend([f_location, "-framework MyFramework"])
    self.cpp_info.exelinkflags = self.cpp_info.sharedlinkflags

13.16 How to collect licenses of dependencies

With the imports feature it is possible to collect the License files from all packages in the dependency graph. Please note that the licenses are artifacts that must exist in the binary packages to be collected, as different binary packages might have different licenses. E.g., A package creator might provide a different license for static or shared linkage with different “License” files if they want to.

Also, we will assume the convention that the package authors will provide a “License” (case not important) file at the root of their packages.

In conanfile.txt we would use the following syntax:

[imports]
., license* -> ./licenses @ folder=True, ignore_case=True

And in conanfile.py we will use the imports() method:

def imports(self):
    self.copy("license*", dst="licenses", folder=True, ignore_case=True)

In both cases, after conan install, it will store all the found License files inside the local licenses folder, which will contain one subfolder per dependency with the license file inside.

13.17 How to extract licenses from headers

Sometimes there is no license file, and you will need to extract the license from a header file, as in the following example:

def package():
    # Extract the License/s from the header to a file
    tmp = tools.load("header.h")
    license_contents = tmp[2:tmp.find("*/", 1)] # The license begins with a
    # C comment /* and ends with */
    tools.save("LICENSE", license_contents)
    # Package it
    self.copy("license*", dst="licenses", ignore_case=True, keep_path=False)
13.18 How to capture package version from SCM: git

The Git() helper from tools can be used to capture data from the Git repo in which the conanfile.py recipe resides, and use it to define the version of the Conan package.

```python
from conans import ConanFile, tools

def get_version():
    git = tools.Git()
    try:
        return "$S$_$S" % (git.get_branch(), git.get_revision())
    except:
        return None

class HelloConan(ConanFile):
    name = "Hello"
    version = get_version()

    def build(self):
        ...
```

In this example, the package created with conan create will be called Hello/branch_commit@user/channel. Note that get_version() returns None if it is not able to get the Git data. This is necessary when the recipe is already in the Conan cache, and the Git repository may not be there. A value of None makes Conan get the version from the metadata.

13.19 How to capture package version from SCM: svn

The SVN() helper from tools can be used to capture data from the subversion repo in which the conanfile.py recipe resides, and use it to define the version of the Conan package.

```python
from conans import ConanFile, tools

def get_svn_version(version):
    try:
        scm = tools.SVN()
        revision = scm.get_revision()
        branch = scm.get_branch()  # Delivers e.g trunk, tags/v1.0.0, branches/my_
        branch = branch.replace("/","_")
        if scm.is_pristine():
            dirty = ""
        else:
            dirty = ".dirty"
        return "$S-$S+$S$s" % (version, revision, branch, dirty)  # e.g. 1.2.0-
    except Exception:
        return None

class HelloLibrary(ConanFile):
    name = "Hello"
    version = get_svn_version("1.2.0")

    def build(self):
        ...
```
In this example, the package created with `conan create` will be called `Hello/generated_version@user/channel`. Note that `get_svn_version()` returns `None` if it is not able to get the subversion data. This is necessary when the recipe is already in the Conan cache, and the subversion repository may not be there. A value of `None` makes Conan get the version from the metadata.

### 13.20 How to capture package version from text or build files

It is common that a library version number would be already encoded in a text file, build scripts, etc. As an example, let’s assume we have the following library layout, and that we want to create a package from it:

```plaintext
conanfile.py
CMakeLists.txt
src
  hello.cpp
  ...
```

The `CMakeLists.txt` will have some variables to define the library version number. For simplicity, let’s also assume that it includes a line such as the following:

```plaintext
cmake_minimum_required(VERSION 2.8)
set(MY_LIBRARY_VERSION 1.2.3) # This is the version we want
add_library(hello src/hello.cpp)
```

Typically, our `conanfile.py` package recipe will include:

```python
class HelloConan(ConanFile):
    name = "Hello"
    version = "1.2.3"
```

This usually requires very little maintenance, and when the CMakeLists version is bumped, so is the `conanfile.py` version. However, if you only want to have to update the `CMakeLists.txt` version, you can extract the version dynamically, using:

```python
from conans import ConanFile
from conans.tools import load
import re

def get_version():
    try:
        content = load("CMakeLists.txt")
        version = re.search(b"set\(MY_LIBRARY_VERSION (.*)\)\", content).group(1)
        return version.strip()
    except Exception as e:
        return None

class HelloConan(ConanFile):
    name = "Hello"
    version = get_version()
```

Even if the `CMakeLists.txt` file is not exported to the local cache, it will still work, as the `get_version()` function returns `None` when it is not found, and then takes the version number from the package metadata (layout).
13.21 How to use Conan as other language package manager

Conan is a generic package manager. In the *getting started* section we saw how to use Conan and manage a C/C++ library, like POCO.

But Conan just provided some tools, related to C/C++ (like some generators and the cpp_info), to offer a better user experience. The general basis of Conan can be used with other programming languages.

Obviously, this does not try to compete with other package managers. Conan is a C and C++ package manager, focused on C and C++ developers. But when we realized that this was possible, we thought it was a good way to showcase its power, simplicity and versatility.

And of course, if you are doing C/C++ and occasionally you need some package from other language in your workflow, as in the Conan package recipes themselves, or for some other tooling, you might find this functionality useful.

13.21.1 Conan: A Go package manager

The source code

You can just clone the following example repository:

```bash
$ git clone https://github.com/conan-community/conan-goserver-example
```

Or, alternatively, manually create the folder and copy the following files inside:

```bash
$ mkdir conan-goserver-example
$ cd conan-goserver-example
$ mkdir src
$ mkdir src/server
```

The files are:

*src/server/main.go* is a small http server that will answer “Hello world!” if we connect to it.

```go
package main

import "github.com/go-martini/martini"

func main() {
  m := martini.Classic()
  m.Get("/", func() string {
    return "Hello world!"
  })
  m.Run()
}
```

Declaring and installing dependencies

Create a *conanfile.txt*, with the following content:

```
[requires]
go-martini/1.0@lasote/stable
```

(continues on next page)
Our project requires a package, go-martini/1.0@lasote/stable, and we indicate that all src contents from all our requirements have to be copied to ./deps/src.

The package go-martini depends on go-inject, so Conan will handle automatically the go-inject dependency.

$ conan install .

This command will download our packages and will copy the contents in the ./deps/src folder.

**Running our server**

Just add the deps folder to GOPATH:

```
# Linux / Macos
$ export GOPATH=${GOPATH}:${PWD}/deps
# Windows
$ SET GOPATH=%GOPATH%;%CD%/deps
```

And run the server:

```
$ cd src/server
$ go run main.go
```

Open your browser and go to localhost:9300

Hello World!

**Generating Go packages**

Creating a Conan package for a Go library is very simple. In a Go project, you compile all the code from sources in the project itself, including all of its dependencies.

So we don’t need to take care of settings at all. Architecture, compiler, operating system, etc. are only relevant for pre-compiled binaries. Source code packages are settings agnostic.

Let’s take a look at the conanfile.py of the go inject library:

```
from conans import ConanFile

class InjectConan(ConanFile):
    name = "go-inject"
    version = "1.0"

    def source(self):
        self.run("git clone https://github.com/codegangsta/inject.git")
        self.run("cd inject && git checkout v1.0-rc1")  # TAG v1.0-rc1

    def package(self):
        self.copy(pattern='*', dst='src/github.com/codegangsta/inject', src="inject", keep_path=True)
```

(continues on next page)
If you have read the *Building a hello world package*, the previous code may look quite simple to you.

We want to pack version 1.0 of the *go inject* library, so the version variable is “1.0”. In the `source()` method, we declare how to obtain the source code of the library, in this case just by cloning the github repository and making a checkout of the `v1.0-rc1` tag. In the `package()` method, we are just copying all the sources to a folder named “src/github.com/codegangsta/inject”.

This way, we can keep importing the library in the same way:

```python
import "github.com/codegangsta/inject"
```

We can export and upload the package to a remote and we are done:

```
$ conan export . lasote/stable  # Or any other user/channel
$ conan upload go-inject/1.0@lasote/stable --all
```

Now look at the *go martini* conanfile:

```
from conans import ConanFile

class InjectConan(ConanFile):
    name = "go-martini"
    version = "1.0"
    requires = 'go-inject/1.0@lasote/stable'

    def source(self):
        self.run("git clone https://github.com/go-martini/martini.git")
        self.run("cd martini && git checkout v1.0")  # TAG v1.0

    def package(self):
        self.copy(pattern='*', dst='src/github.com/go-martini/martini', src="martini ...", keep_path=True)
```

It is very similar. The only difference is the `requires` variable. It defines the `go-inject/1.0@lasote/stable` library, as a requirement.

```
$ conan export . lasote/stable  # Or any other user/channel
$ conan upload go-martini/1.0@lasote/stable --all
```

Now we are able to use them easily and without the problems of versioning with github checkouts.

### 13.21.2 Conan: A Python package manager

Conan is a C and C++ package manager, and to deal with the vast variability of C and C++ build systems, compilers, configurations, etc., it was designed to be extremely flexible, to allow users the freedom to configure builds in virtually any manner required. This is one of the reasons to use Python as the scripting language for Conan package recipes.

With this flexibility, Conan is able to do very different tasks: package Visual Studio modules, *package Go code*, build packages from sources or from binaries retrieved from elsewhere, etc.

*Python code can be reused and packaged with Conan* to share functionalities or tools among *conanfile.py* files. Here we can see a full example of Conan as a Python package manager.
A full Python and C/C++ package manager

The real utility of this is that Conan is a C and C++ package manager. So, for example, you are able to create a Python package that wraps the functionality of the Poco C++ library. Poco itself has transitive (C/C++) dependencies, but they are already handled by Conan. Furthermore, a very interesting thing is that nothing has to be done in advance for that library, thanks to useful tools such as pybind11, that lets you easily create Python bindings.

So let’s build a package with the following files:

- **conanfile.py**: The package recipe.
- **__init__.py**: A required file which should remain blank.
- **pypoco.cpp**: The C++ code with the pybind11 wrapper for Poco that generates a Python extension (a shared library that can be imported from Python).
- **CMakeLists.txt**: The CMake build file that is able to compile pypoco.cpp into a Python extension (pypoco.pyd in Windows, pypoco.so in Linux)
- **poco.py**: A Python file that makes use of the pypoco Python binary extension built with pypoco.cpp.
- **test_package/conanfile.py**: A test consumer “convenience” recipe to create and test the package.

The **pypoco.cpp** file can be coded easily thanks to the elegant pybind11 library:

```cpp
#include <pybind11/pybind11.h>
#include "Poco/Random.h"

using Poco::Random;
namespace py = pybind11;

PYBIND11_PLUGIN(pypoco) {
    py::module m("pypoco", "pybind11 example plugin");
    py::class_<Random>({m, "Random"})
        .def(py::init<>())
        .def("nextFloat", &Random::nextFloat);
    return m.ptr();
}
```

And the **poco.py** file is straightforward:

```python
import sys
import pypoco

def random_float():
    r = pypoco.Random()
    return r.nextFloat()
```

The **conanfile.py** is a bit longer, but is still quite easy to understand:

```python
from conans import ConanFile, tools, CMake

class PocoPyReuseConan(ConanFile):
    name = "PocoPy"
```

(continues on next page)
The recipe now declares 2 requires that will be used to create the binary extension: the Poco library and the pybind11 library.

As we are actually building C++ code, there are a few important things that we need:

- Input settings that define the OS, compiler, version and architecture we are using to build our extension. This is necessary because the binary we are building must match the architecture of the Python interpreter that we will be using.

- The build() method is actually used to invoke CMake. You may see that we had to hardcode the Python path in the example, as the CMakeLists.txt call to find_package(PythonLibs) didn’t find my Python installation in C:/Python27, even though that is a standard path. I have also added the cmake generator to be able to easily use the declared requires build information inside my CMakeLists.txt.

- The CMakeLists.txt is not posted here, but is basically the one used in the pybind11 example with just 2 lines to include the cmake file generated by Conan for dependencies. It can be inspected in the GitHub repo.

- Note that we are using Python 2.7 as an input option. If necessary, more options for other interpreters/architectures could be easily provided, as well as avoiding the hardcoded paths. Even the Python interpreter itself could be packaged in a Conan package.

The above recipe will generate a different binary for different compilers or versions. As the binary is being wrapped by Python, we could avoid this and use the same binary for different setups, modifying this behavior with the conan_info() method.

Now, the first invocation of conan install will retrieve the dependencies and build the package. The next invocation will use the cached binaries and be much faster. Note how we have to specify -s arch=x86 to match the architecture of the Python interpreter to be used, in our case, 32 bits.
The output of the `conan install` command also shows us the dependencies that are being pulled:

```
Requirements
OpenSSL/1.0.2l@conan/stable from conan.io
Poco/1.9.0@pocoproject/stable from conan.io
PocoPy/0.1@memsharded/testing from local
pybind11/any@memsharded/stable from conan.io
zlib/1.2.11@conan/stable from conan.io
```

This is one of the great advantages of using Conan for this task, because by depending on Poco, other C and C++ transitive dependencies are retrieved and used in the application.

For a deeper look into the code of these examples, please refer to this github repo. The above examples and code have only been tested on Win10, VS14u2, but may work on other configurations with little or no extra work.

### 13.22 How to manage SSL (TLS) certificates

#### 13.22.1 Server certificate validation

By default, when a remote is added, if the URL schema is `https`, the Conan client will verify the certificate using a list of authorities declared in the `cacert.pem` file located in the Conan home (`~/.conan`).

If you have a self signed certificate (not signed by any authority) you have two options:

- Use the `conan remote` command to disable the SSL verification.
- Append your server `crt` file to the `cacert.pem` file.

#### 13.22.2 Client certificates

If your server is requiring client certificates to validate a connection from a Conan client, you need to create two files in the Conan home directory (default `~/.conan`):

- A file `client.crt` with the client certificate.
- A file `client.key` with the private key.

**Note:** You can create only the `client.crt` file containing both the certificate and the private key concatenated and not create the `client.key`.

If you are a familiar with the `curl` tool, this mechanism is similar to specify the `--cert`/`--key` parameters.

### 13.23 How to check the version of the Conan client inside a conanfile

Sometimes it might be useful to check the Conan version that is running in that moment your recipe. Although we consider conan-center recipes only forward compatible, this kind of check makes sense to update them so they can maintain compatibility with old versions of Conan.

Let’s have a look at a basic example of this:
Listing 17: conanfile.py

```python
from conans import ConanFile, CMake, __version__ as conan_version
from conans.model.version import Version

class MyLibraryConan(ConanFile):
    name = "mylibrary"
    version = "1.0"

    def build(self):
        if conan_version < Version("0.29"):
            cmake = CMake(self.settings)
        else:
            cmake = CMake(self)

        ...
```

Here it checks the Conan version to maintain compatibility of the CMake build helper for versions lower than Conan 0.29. It also uses the internal `Version()` class to perform the semver comparison in the if clause.

You can find a real example of this in the `mingw_installer`. Here you have the interesting part of the recipe:

Listing 18: conanfile.py

```python
from conans import ConanFile, tools, __version__ as conan_version
from conans.model.version import Version

class MingwInstallerConan(ConanFile):
    name = "mingw_installer"
    version = "1.0"
    license = "http://www.mingw.org/license"
    url = "http://github.com/conan-community/conan-mingw-installer"

    if conan_version < Version("0.99"):
        os_name = "os"
        arch_name = "arch"
    else:
        os_name = "os_build"
        arch_name = "arch_build"

    settings = {os_name: ["Windows"],
                arch_name: ["x86", "x86_64"],
                "compiler": {"gcc": {"version": None,
                                     "libcxx": ["libstdc++", "libstdc++11"],
                                     "threads": ["posix", "win32"],
                                     "exception": ["dwarf2", "sjlj", "seh"]}}}

    ...
```

You can see here the `mingw_installer` recipe uses new settings `os_build` and `arch_build` since Conan 1.0 as those are the right ones for installer packages. However, it also keeps the old settings so as not to break the recipe for old version, using normal `os` and `arch`.

As said before, this is useful to maintain recipe compatibility with older Conan versions but remember that since Conan 1.0 there should not be any breaking changes.

13.23. How to check the version of the Conan client inside a conanfile 255
13.24 Use a generic CI with Conan and Artifactory

13.24.1 Uploading the BuildInfo

If you are using Jenkins with Conan and Artifactory, along with the Jenkins Artifactory Plugin, any Conan package downloaded or uploaded during your build will be automatically recorded in the BuildInfo json file, that will be automatically uploaded to the specified Artifactory instance.

However, you can gather and upload that information using other CI infrastructure with the following steps:

1. Before calling Conan the first time in your build, set the environment variable CONAN_TRACE_FILE to a file path. The generated file will contain the BuildInfo json.

2. You also need to create the artifacts.properties file in your Conan home containing the build information. All this properties will be automatically associated to all the published artifacts.

3. Call Conan as many times as you need. For example, if you are testing a Conan package and uploading it at the end, you will run something similar to:

   $ conan create . user/stable # Will retrieve the dependencies and create the package
   $ conan upload mypackage/1.0@user/stable -r artifactory

4. Call the command conan_build_info passing the path to the generated Conan traces file and a parameter --output to indicate the output file. You can also, delete the traces.log file otherwise while the CONAN_TRACE_FILE is present, any Conan command will keep appending actions.

   $ conan_build_info /tmp/traces.log --output /tmp/build_info.json
   $ rm /tmp/traces.log

5. Edit the build_info.json file to append name (build name), number (build number) and the started (started date) and any other field that you need according to the Build Info json format.

   The started field has to be in the format: yyyy-MM-dd'T'HH:mm:ss.SSSZ

   To edit the file you can import the json file using the programming language you are using in your framework, groovy, java, python...

6. Push the json file to Artifactory, using the REST-API:

   curl -X PUT -u<username>:<password> -H "Content-type: application/json" -T /tmp/build_
   →info.json "http://host:8081/artifactory/api/build"
General information about the commands, configuration files, etc.

Contents:

14.1 Commands

14.1.1 Consumer commands

Commands related with the installation and usage of Conan packages:

```
conan install
```

```
$ conan install [-h] [-g GENERATOR] [-if INSTALL_FOLDER] [-m [MANIFESTS]]
[mi [MANIFESTS_INTERACTIVE]] [-v [VERIFY]]
[--no-imports] [-j JSON] [-b [BUILD]] [-e ENV]
[-o OPTIONS] [-pr PROFILE] [-r REMOTE] [-s SETTINGS] [-u]
path_or_reference [reference]
```

Installs the requirements specified in a recipe (conanfile.py or conanfile.txt).

It can also be used to install a concrete package specifying a reference. If any requirement is not found in the local cache, it will retrieve the recipe from a remote, looking for it sequentially in the configured remotes. When the recipes have been downloaded it will try to download a binary package matching the specified settings, only from the remote from which the recipe was retrieved. If no binary package is found, it can be build from sources using the ‘–build’ option. When the package is installed, Conan will write the files for the specified generators.

**positional arguments:**
- `path_or_reference` Path to a folder containing a recipe (conanfile.py or conanfile.txt) or to a recipe file. e.g., ./my_project/conanfile.txt. It could also be a reference
- `reference` Reference for the conanfile path of the first argument: user/channel, version@user/channel or pkg/version@user/channel (if name or version declared in conanfile.py, they should match)

**optional arguments:**
- `--help` show this help message and exit
- `-g GENERATOR, --generator GENERATOR` Generators to use

(continues on next page)
-if INSTALL_FOLDER, --install-folder INSTALL_FOLDER
   Use this directory as the directory where to put the
   generator files. e.g., conaninfo/conanbuildinfo.txt

-m [MANIFESTS], --manifests [MANIFESTS]
   Install dependencies manifests in folder for later
   verify. Default folder is .conan_manifests, but can be
   changed

-mi [MANIFESTS_INTERACTIVE], --manifests-interactive [MANIFESTS_INTERACTIVE]
   Install dependencies manifests in folder for later
   verify, asking user for confirmation. Default folder
   is .conan_manifests, but can be changed

-v [VERIFY], --verify [VERIFY]
   Verify dependencies manifests against stored ones

--no-imports
   Install specified packages but avoid running imports

-j JSON, --json JSON
   Path to a json file where the install information will
   be written

-b [BUILD], --build [BUILD]
   Optional, use it to choose if you want to build from
   sources: --build Build all from sources, do not use
   binary packages. --build=never Never build, use binary
   packages or fail if a binary package is not found.
   --build=missing Build from code if a binary package is
   not found. --build=cascade Will build from code all
   the nodes with some dependency being built (for any
   reason). Can be used together with any other build
   policy. Useful to make sure that any new change
   introduced in a dependency is incorporated by building
   again the package. --build=outdated Build from code if
   the binary is not built with the current recipe or
   when missing binary package. --build=[pattern] Build
   always these packages from source, but never build the
   others. Allows multiple --build parameters. 'pattern' is
   a fnmatch file pattern of a package reference.
   Default behavior: If you don't specify anything, it
   will be similar to '--build=never', but package
   recipes can override it with their 'build_policy'
   attribute in the conanfile.py.

-e ENV, --env ENV
   Environment variables that will be set during the
   package build, -e CXX=/usr/bin/clang++

-o OPTIONS, --options OPTIONS
   Define options values, e.g., -o Pkg:with_qt=true

-pr PROFILE, --profile PROFILE
   Apply the specified profile to the install command

-r REMOTE, --remote REMOTE
   Look in the specified remote server

-s SETTINGS, --settings SETTINGS
   Settings to build the package, overwriting the
defaults. e.g., -s compiler=gcc

-u, --update
   Check updates exist from upstream remotes

**conan install** executes methods of a *conanfile.py* in the following order:

1. config_options()
2. configure()
3. requirements()
4. package_id()
5. package_info()
6. deploy()

Note this describes the process of installing a pre-built binary package. If the package has to be built, `conan install --build` executes the following:

1. config_options()
2. configure()
3. requirements()
4. package_id()
5. build_requirements()
6. build_id()
7. system_requirements()
8. source()
9. imports()
10. build()
11. package()
12. package_info()
13. deploy()

Examples

- Install a package requirement from a `conanfile.txt`, saved in your current directory with one option and setting (other settings will be defaulted as defined in `<userhome>/.conan/profiles/default`):

  $ conan install . -o PkgName:use_debug_mode=on -s compiler=clang

- Install the requirements defined in a `conanfile.py` file in your current directory, with the default settings in default profile `<userhome>/.conan/profiles/default`, and specifying the version, user and channel (as they might be used in the recipe):

  ```python
class Pkg(ConanFile):
    name = "mypkg"
    # see, no version defined!
    def requirements(self):
      # this trick allow to depend on packages on your same user/channel
      self.requires("dep/0.3@%s/%s" % (self.user, self.channel))

    def build(self):
      if self.version == "myversion":
        # something specific for this version of the package.

  $ conan install . myversion@someuser/somechannel
  ```

Those values are cached in a file, so later calls to local commands like `conan build` can find and use this version, user and channel data.

- Install the `OpenCV/2.4.10@lasote/testing` reference with its default options and default settings from `<userhome>/.conan/profiles/default`:

14.1. Commands
conan Documentation, Release 1.16.0

$ conan install opencv/2.4.10@lasote/testing

• Install the OpenCV/2.4.10@lasote/testing reference updating the recipe and the binary package if new up-stream versions are available:

$ conan install opencv/2.4.10@lasote/testing --update

build options

Both the conan install and create commands have options to specify whether conan should try to build things or not:

• --build=never: This is the default option. It is not necessary to write it explicitly. Conan will not try to build packages when the requested configuration does not match, in which case it will throw an error.

• --build=missing: Conan will try to build from source, all packages of which the requested configuration was not found on any of the active remotes.

• --build=outdated: Conan will try to build from code if the binary is not built with the current recipe or when missing binary package.

• --build=[pattern]: A fnmatch case-sensitive pattern of a package reference or only the package name. Conan will force the build of the packages whose reference matches the given pattern. Several patterns can be specified, chaining multiple options:
  - e.g., --build=pattern1 --build=pattern2 can be used to specify more than one pattern.
  - e.g., --build=zlib will match any package named zlib (same as zlib/*).
  - e.g., --build=z*@conan/stable will match any package starting with z with conan/stable as user/channel.

• --build: Always build everything from source. Produces a clean re-build of all packages and transitively dependent packages

env variables

With the -e parameters you can define:

• Global environment variables (-e SOME_VAR="SOME_VALUE"). These variables will be defined before the build step in all the packages and will be cleaned after the build execution.

• Specific package environment variables (-e zlib:SOME_VAR="SOME_VALUE"). These variables will be defined only in the specified packages (e.g., zlib).

You can specify this variables not only for your direct requires but for any package in the dependency graph.

If you want to define an environment variable but you want to append the variables declared in your requirements you can use the [] syntax:

$ conan install . -e PYTHONPATH=/[other/path]

This way the first entry in the PYTHONPATH variable will be /other/path but the PYTHONPATH values declared in the requirements of the project will be appended at the end using the system path separator.
settings

With the `-s` parameters you can define:

- Global settings (`-s compiler="Visual Studio"`). Will apply to all the requires.
- Specific package settings (`-s zlib:compiler="MinGW"`). Those settings will be applied only to the specified packages.

You can specify custom settings not only for your direct requires but for any package in the dependency graph.

options

With the `-o` parameters you can only define specific package options.

```bash
$ conan install . -o zlib:shared=True
$ conan install . -o zlib:shared=True -o bzip2:option=132
# you can also apply the same options to many packages with wildcards:
$ conan install . -o *:shared=True
```

Note: You can use profiles files to create predefined sets of settings, options and environment variables.

reference

An optional positional argument, if used the first argument should be a path. If the reference specifies name and/or version, and they are also declared in the `conanfile.py`, they should match, otherwise, an error will be raised.

```bash
$ conan install . # OK, user and channel will be None
$ conan install . user/testing # OK
$ conan install . version@user/testing # OK
$ conan install . pkg/version@user/testing # Error, first arg is not a path
$ conan install pkg/version@user/testing user/channel # Error, first arg is not a path
```

conan config

```bash
$ conan config [-h] {rm,set,get,install} ...
```

Manages Conan configuration.

Used to edit conan.conf, or install config files.

<table>
<thead>
<tr>
<th>positional arguments:</th>
<th>sub-command help</th>
</tr>
</thead>
<tbody>
<tr>
<td>{rm,set,get,install}</td>
<td>Remove an existing config element</td>
</tr>
<tr>
<td>rm</td>
<td>Set a value for a configuration item</td>
</tr>
<tr>
<td>set</td>
<td>Get the value of configuration item</td>
</tr>
<tr>
<td>install</td>
<td>install a full configuration from a local or remote zip file</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>optional arguments:</th>
<th>show this help message and exit</th>
</tr>
</thead>
<tbody>
<tr>
<td>-h, --help</td>
<td></td>
</tr>
</tbody>
</table>
Examples

• Change the logging level to 10:

$ conan config set log.level=10

• Get the logging level:

$ conan config get log.level
$> 10

conan config install

usage: conan config install [-h] [--verify-ssl [VERIFY_SSL]] [--type {git}]
                  [--args ARGS] [-sf SOURCE_FOLDER]
                  [-tf TARGET_FOLDER]
                  [item]

positional arguments:
  item     git repository, local folder or zip file (local or http) where the configuration is stored

optional arguments:
  -h, --help       show this help message and exit
  --verify-ssl [VERIFY_SSL]
                   Verify SSL connection when downloading file
  --type {git}, -t {git}
                   Type of remote config
  --args ARGS, -a 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` is intended to share the Conan client configuration. For example, in a company or organization, is important to have common `settings.yml`, `profiles`, etc.

It can get its configuration files from a local or remote zip file, from a local directory or from a git repository. It then installs the files in the local Conan configuration.

The configuration may contain all or a subset of the allowed configuration files. Only the files that are present will be replaced. The only exception is the `conan.conf` file for which only the variables declared will be installed, leaving the other variables unchanged.

This means for example that `profiles` and `hooks` files will be overwritten if already present, but no profile or hook file that the user has in the local machine will be deleted.

All the configuration files will be copied to the Conan home directory. These are the special files and the rules applied to merge them:

<table>
<thead>
<tr>
<th>File</th>
<th>How it is applied</th>
</tr>
</thead>
<tbody>
<tr>
<td>profiles/MyProfile</td>
<td>Overrides the local ~/.conan/profiles/MyProfile if already exists</td>
</tr>
<tr>
<td>settings.yml</td>
<td>Overrides the local ~/.conan/settings.yml</td>
</tr>
<tr>
<td>remotes.txt</td>
<td>Overrides remotes. Will remove remotes that are not present in file</td>
</tr>
<tr>
<td>config/conan.conf</td>
<td>Merges the variables, overriding only the declared variables</td>
</tr>
<tr>
<td>hooks/my_hook.py</td>
<td>Overrides the local ~/.conan/hooks/my_hook.py if already exists</td>
</tr>
</tbody>
</table>
The file `remotes.txt` is the only file listed above which does not have a direct counterpart in the `~/.conan` folder. Its format is a list of entries, one on each line, with the form of

```
[remote name] [remote url] [bool]
```

where `[bool]` (either True or False) indicates whether SSL should be used to verify that remote. The remote definitions can be found in the `registry.txt/registry.json` files and they provide a helpful starting point when writing the `remotes.txt` to be packaged in a Conan client configuration.

**Important:** The local cache `registry.txt/registry.json` file contains the remotes definitions as well as the mapping of installed packages from remotes. Sharing the complete contents of this file via this command is not recommended as this records the status of the local cache, which may be different from one machine to another.

**Note:** During the installation, Conan skips any file with the name `README.md` or `LICENSE.txt`.

The `conan config install <item>` calls are stored in a `config_install.json` file in the Conan local cache. That allows to issue a `conan config install` command, without arguments, to iterate over the cached configurations, executing them again (updating).

**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
  # Later on, execute again the previous configurations cached:
  $ conan config install
  ```

  It’s not needed to specify any argument, it will iterate previously stored configurations in `config_install.json`, executing them again.

- 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 the configuration from a local path:

  $ conan config install /path/to/some/config.zip

conan get


Gets a file or list a directory of a given reference or package.

positional arguments:
  reference  Recipe reference or package reference e.g., 'MyPackage/1.2@user/channel', 'MyPackage/1.2@user/channel:af7901d8bd6edef21d086181aa1c495c25a17b137'
  path      Path to the file or directory. If not specified will get the conanfile if only a reference is specified and a conaninfo.txt file contents if the package is also specified

optional arguments:
  -h, --help     show this help message and exit
  -p PACKAGE, --package PACKAGE
                  Package ID [DEPRECATED: use full reference instead]
  -r REMOTE, --remote REMOTE
                  Get from this specific remote
  -raw, --raw
                  Do not decorate the text

Examples:

- Print the conanfile.py from a remote package:

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

- List the files for a local package recipe:

  $ conan get zlib/1.2.11@conan/stable .

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

- Print a file from a recipe folder:

  $ conan get zlib/1.2.11@conan/stable conanmanifest.txt

- Print the conaninfo.txt file for a binary package:

  $ conan get zlib/1.2.11@conan/stable:2144f833c251030c3cf6c4354ae0e38607a909
[settings]
arch=x86_64
build_type=Release
compiler=apple-clang
compiler.version=8.1
os=Macos

[requires]

[options]
#

- List the files from a binary package in a remote:

```bash
$ conan get zlib/1.2.11@conan/stable:2144f833c251030c3cf61c4354ae0e38607a909 | ˓→ r conan-center
Listing directory '.':
conan_package.tgz
conaninfo.txt
conanmanifest.txt
```

### conan info

```bash
```

Gets information about the dependency graph of a recipe.

It can be used with a recipe or a reference for any existing package in your local cache.

Positional arguments:

- **path_or_reference**: Path to a folder containing a recipe (conanfile.py or conanfile.txt) or to a recipe file. E.g., ./my_project/conanfile.txt. It could also be a reference.

Optional arguments:

- **-h, --help**: show this help message and exit
- **--paths**: Show package paths in local cache
- **--build-order BUILD_ORDER**: given a modified reference, return an ordered list to build (CI)
- **-g GRAPH, --graph GRAPH**: Creates file with project dependencies graph. It will generate a DOT or HTML file depending on the filename extension
- **-if INSTALL_FOLDER, --install-folder INSTALL_FOLDER**: local folder containing the conaninfo.txt and conanbuildinfo.txt files (from a previous conan install execution). Defaulted to current folder,
unless --profile, -s or -o is specified. If you specify both install-folder and any setting/option it will raise an error.

`-j [JSON], --json [JSON]`  
Path to a json file where the information will be written

`-n ONLY, --only ONLY`  
Show only the specified fields: "id", "build_id", "remote", "url", "license", "requires", "update", "required", "date", "author", "None". '--paths' information can also be filtered with options "export_folder", "build_folder", "package_folder", "source_folder". Use 'only None' to show only references.

`--package-filter [PACKAGE_FILTER]`  
Print information only for packages that match the filter pattern e.g., MyPackage/1.2@user/channel or MyPackage*

`-db [DRY_BUILD], --dry-build [DRY_BUILD]`  
Apply the --build argument to output the information, as it would be done by the install command

`-b [BUILD], --build [BUILD]`  
Given a build policy, return an ordered list of packages that would be built from sources during the install command

`-e ENV, --env ENV`  
Environment variables that will be set during the package build, -e CXX=/usr/bin/clang++

`-o OPTIONS, --options OPTIONS`  
Define options values, e.g., -o Pkg:with_qt=true

`-pr PROFILE, --profile PROFILE`  
Apply the specified profile to the install command

`-r REMOTE, --remote REMOTE`  
Look in the specified remote server

`-s SETTINGS, --settings SETTINGS`  
Settings to build the package, overwriting the defaults. e.g., -s compiler=gcc

`-u, --update`  
Check updates exist from upstream remotes

**Examples:**

```
$ conan info .
$ conan info myproject_folder
$ conan info myproject_folder/conanfile.py
$ conan info Hello/1.0@user/channel
```

The output will look like:

```
Dependency/0.1@user/channel
  ID: 5ab84d6acfe1f23c4fae0ab88f26e3a396351ac9
  BuildID: None
  Remote: None
  URL: http://...
  License: MIT
  Updates: Version not checked
  Creation date: 2017-10-31 14:45:34
  Required by:
    Hello/1.0@user/channel
```

(continues on next page)
conan info builds the complete dependency graph, like conan install does. The main difference is that it doesn’t try to install or build the binaries, but the package recipes will be retrieved from remotes if necessary.

It is very important to note, that the info command outputs the dependency graph for a given configuration (settings, options), as the dependency graph can be different for different configurations. Then, the input to the conan info command is the same as conan install, the configuration can be specified directly with settings and options, or using profiles.

Also, if you did a previous conan install with a specific configuration, or maybe different installs with different configurations, you can reuse that information with the --install-folder argument:

```
$ # dir with a conanfile.txt
$ mkdir build_release && cd build_release
$ conan install .. --profile=gcc54release
$ cd .. && mkdir build_debug && cd build_debug
$ conan install .. --profile=gcc54debug
$ cd ..
$ conan info . --install-folder=build_release
> info for the release dependency graph install
$ conan info . --install-folder=build_debug
> info for the debug dependency graph install
```

It is possible to use the conan info command to extract useful information for Continuous Integration systems. More precisely, it has the --build-order, -bo option, that will produce a machine-readable output with an ordered list of package references, in the order they should be built. E.g., let’s assume that we have a project that depends on Boost and Poco, which in turn depends on OpenSSL and zlib transitively. So we can query our project with a reference that has changed (most likely due to a git push on that package):

```
$ conan info . --bo zlib/1.2.11@conan/stable
[zlib/1.2.11@conan/stable], [OpenSSL/1.0.21@conan/stable], [Boost/1.60.0@lasote/
→stable, Poco/1.7.8p3@pocoproject/stable]
```

Note the result is a list of lists. When there is more than one element in one of the lists, it means that they are decoupled projects and they can be built in parallel by the CI system.

You can also specify the --build-order=ALL argument, if you want just to compute the whole dependency graph build order

```
$ conan info . --build-order=ALL
> [zlib/1.2.11@conan/stable], [OpenSSL/1.0.21@conan/stable], [Boost/1.60.0@lasote/
→stable, Poco/1.7.8p3@pocoproject/stable]
```

Also you can get a list of nodes that would be built (simulation) in an install command specifying a build policy with the --build parameter.
E.g., if I try to install Boost/1.60.0@lasote/stable recipe with --build missing build policy and arch=x86, which libraries will be built?

```bash
$ conan info Boost/1.60.0@lasote/stable --build missing -s arch=x86
bzip2/1.0.6@lasote/stable, zlib/1.2.8@lasote/stable, Boost/1.60.0@lasote/stable
```

You can generate a graph of your dependencies, in dot or html formats:

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

The generated html output contains links to third party resources, the vis.js library (2 files: vis.min.js, vis.min.css). By default they are retrieved from cloudflare. However, for environments without internet connection, these files could be also used from the local cache and installed with conan config install by putting those files in the root of the configuration folder:

- **vis.min.js**: Default link to “https://cdnjs.cloudflare.com/ajax/libs/vis/4.18.1/vis.min.js”
- **vis.min.css**: Default link to “https://cdnjs.cloudflare.com/ajax/libs/vis/4.18.1/vis.min.css”

It is not necessary to modify the generated html file. Conan will automatically use the local paths to the cache files if present, or the internet ones if not.

**conan search**

```bash
```

Searches package recipes and binaries in the local cache or in a remote.

If you provide a pattern, then it will search for existing package recipes matching it. If a full reference is provided (pkg/0.1@user/channel) then the existing binary packages for that reference will be displayed. If no remote is specified, the search will be done in the local cache. Search is case sensitive, exact case has to be used. For case insensitive file systems, like Windows, case sensitive search can be forced with ‘--case-sensitive’.
optional arguments:
- h, --help show this help message and exit
- o, --outdated Show only outdated from recipe packages. This flag can only be used with a reference
- q QUERY, --query QUERY Packages query: 'os=Windows AND (arch=x86 OR compiler=gcc)'. The 'pattern_or_reference' parameter has to be a reference: MyPackage/1.2@user/channel
- r REMOTE, --remote REMOTE Remote to search in. '-r all' searches all remotes
--case-sensitive Make a case-sensitive search. Use it to guarantee case-sensitive search in Windows or other case-insensitive file systems
--raw Print just the list of recipes
--table TABLE Outputs html file with a table of binaries. Only valid for a reference search
-j JSON, --json JSON json file path where the search information will be written to
-rev, --revisions Get a list of revisions for a reference or a package reference.

Examples

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

To search for recipes in all defined remotes use --all (this is only valid for searching recipes, not binaries):

```
$ conan search zlib/* -r=all
```

If you use instead the full package recipe reference, you can explore the binaries existing for that recipe, also in a remote or in the local conan cache:

```
$ conan search Boost/1.60.0@lasote/stable
```

A query syntax is allowed to look for specific binaries, you can use AND and OR operators and parenthesis, with settings and also options.

```
$ conan search Boost/1.60.0@lasote/stable -q arch=x86_64
$ conan search Boost/1.60.0@lasote/stable -q "(arch=x86_64 OR arch=ARM) AND (build_\_\_type=Release OR os=Windows)"
```

If you specify a query filter for a setting and the package recipe is not restricted by this setting, Conan won’t find the packages. e.g:

```python
class MyRecipe(ConanFile):
    settings="arch"
```

```
$ conan search MyRecipe/1.0@lasote/stable -q os=Windows
```

The query above won’t find the MyRecipe binary packages (because the recipe doesn’t declare “os” as a setting) unless you specify the None value:

```
$ conan search MyRecipe/1.0@lasote/stable -q os=None
```
You can generate a table for all binaries from a given recipe with the `--table` option:

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

### zlib/1.2.11@conan/stable

![Table of zlib binaries](image)

- Selected: 52365c918e417d0f0485a367c434eb3c362d08

#### Legend

- Outdated from recipe
- Updated
- Non-existing

Search the local revision for a local cache recipe:

```
$ conan search lib/1.0@conan/testing --revisions
$ Revisions for 'lib/1.0@conan/testing':
  a55e3b054fdbf4e2c6f10e955da69502 (2019-03-05 16:37:27 UTC)
```

Search the remote revisions in a server:

```
$ conan search lib/1.0@conan/testing --revisions -r=myremote
Revisions for 'lib/1.0@conan/testing' at remote 'myremote':
  78fcef25a1eaeec5fca9bf08624c561 (2019-03-05 16:37:27 UTC)
  f3367e0e7d170a12abccb175f9e5f97 (2019-03-05 16:37:27 UTC)
```

### 14.1.2 Creator commands

Commands related to the creation of Conan recipes and packages:
**conan create**

```bash
```

path reference

**Builds a binary package for a recipe (conanfile.py).**

Uses the specified configuration in a profile or in -s settings, -o options etc. If a ‘test_package’ folder (the name can be configured with -tf) is found, the command will run the consumer project to ensure that the package has been created correctly. Check ‘conan test’ command to know more about ‘test_folder’ project.

**Positional arguments:**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>path</td>
<td>Path to a folder containing a conanfile.py or to a recipe file e.g., my_folder/conanfile.py</td>
</tr>
<tr>
<td>reference</td>
<td>user/channel, version@user/channel or pkg/version@user/channel (if name or version declared in conanfile.py, they should match)</td>
</tr>
</tbody>
</table>

**Optional arguments:**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-h, --help</td>
<td>show this help message and exit</td>
</tr>
<tr>
<td>-j JSON, --json JSON</td>
<td>json file path where the install information will be written to</td>
</tr>
<tr>
<td>-k, -ks, --keep-source</td>
<td>Do not remove the source folder in local cache, even if the recipe changed. Use this for testing purposes only</td>
</tr>
<tr>
<td>-kb, --keep-build</td>
<td>Do not remove the build folder in local cache. Implies --keep-source. Use this for testing purposes only</td>
</tr>
<tr>
<td>-ne, --not-export</td>
<td>Do not export the conanfile.py</td>
</tr>
<tr>
<td>-tbf TEST_BUILD_FOLDER, --test-build-folder TEST_BUILD_FOLDER</td>
<td>Working directory for the build of the test project.</td>
</tr>
<tr>
<td>-tf TEST_FOLDER, --test-folder TEST_FOLDER</td>
<td>Alternative test folder name. By default it is &quot;test_package&quot;. Use &quot;None&quot; to skip the test stage</td>
</tr>
<tr>
<td>-m [MANIFESTS], --manifests [MANIFESTS]</td>
<td>Install dependencies manifests in folder for later verify. Default folder is .conan_manifests, but can be changed</td>
</tr>
<tr>
<td>-mi [MANIFESTS_INTERACTIVE], --manifests-interactive [MANIFESTS_INTERACTIVE]</td>
<td>Install dependencies manifests in folder for later verify, asking user for confirmation. Default folder is .conan_manifests, but can be changed</td>
</tr>
<tr>
<td>-v [VERIFY], --verify [VERIFY]</td>
<td>Verify dependencies manifests against stored ones</td>
</tr>
<tr>
<td>-b [BUILD], --build [BUILD]</td>
<td>Optional, use it to choose if you want to build from sources: --build Build all from sources, do not use binary packages. --build=never Never build, use binary packages or fail if a binary package is not found. --build=missing Build from code if a binary package is not found. --build=cascade Will build from code all the nodes with some dependency being built (for any reason). Can be used together with any other build policy. Useful to make sure that any new change</td>
</tr>
</tbody>
</table>

(continues on next page)
introduced in a dependency is incorporated by building again the package. --build=outdated Build from code if the binary is not built with the current recipe or when missing binary package. --build=[pattern] Build always these packages from source, but never build the others. Allows multiple --build parameters. 'pattern' is a fnmatch file pattern of a package reference. Default behavior: If you don't specify anything, it will be similar to '---build=never', but package recipes can override it with their 'build_policy' attribute in the conanfile.py.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-e ENV, --env ENV</td>
<td>Environment variables that will be set during the package build, -e CXX=/usr/bin/clang++</td>
</tr>
<tr>
<td>-o OPTIONS, --options OPTIONS</td>
<td>Define options values, e.g., -o Pkg:with_qt=true</td>
</tr>
<tr>
<td>-pr PROFILE, --profile PROFILE</td>
<td>Apply the specified profile to the install command</td>
</tr>
<tr>
<td>-r REMOTE, --remote REMOTE</td>
<td>Look in the specified remote server</td>
</tr>
<tr>
<td>-s SETTINGS, --settings SETTINGS</td>
<td>Settings to build the package, overwriting the defaults. e.g., -s compiler=gcc</td>
</tr>
<tr>
<td>-u, --update</td>
<td>Check updates exist from upstream remotes</td>
</tr>
</tbody>
</table>

This is the recommended way to create packages.

The reference field can be:

- A complete package reference: pkg/version@user/channel. In this case, the recipe doesn’t need to declare the name or the version. If the recipe declares them, they should match the provided values in the command line.

- The user and channel: user/channel. The command will assume that the name and version are provided by the recipe.

- The version, user and channel: version@user/channel. The recipe must provide the name, and if it does provide the version, it should match the command line one.

**conan create** . demo/testing is equivalent to:

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

**Tip:** Sometimes you need to skip/disable test stage to avoid a failure while creating the package, i.e: when you are cross compiling libraries and target code cannot be executed in current host platform. In that case you can skip/disable the test package stage:

```bash
$ conan create . demo/testing --test-folder=None
```

**conan create** executes methods of a conanfile.py in the following order:

1. config_options()
2. configure()
3. requirements()
4. package_id()
5. build_requirements()
6. build_id()
7. system_requirements()
8. source()
9. imports()
10. build()
11. package()
12. package_info()

In case of installing a pre-built binary, steps from 5 to 11 will be skipped. Note that deploy() method is only used in conan install.

**conan export**

```bash
$ conan export [-h] [-k] path reference
```

Copies the recipe (conanfile.py & associated files) to your local cache.

Use the ‘reference’ param to specify a user and channel where to export it. Once the recipe is in the local cache it can be shared, reused and to any remote with the ‘conan upload’ command.

### positional arguments:

- **path**
  
  Path to a folder containing a conanfile.py or to a recipe file e.g., my_folder/conanfile.py

- **reference**
  
  user/channel, or Pkg/version@user/channel (if name and version are not declared in the conanfile.py)

### optional arguments:

- **-h, --help**
  
  show this help message and exit

- **-k, --keep-source**
  
  Do not remove the source folder in local cache, even if the recipe changed. Use this for testing purposes only

The reference field can be:

- A complete package reference: pkg/version@user/channel. In this case, the recipe doesn’t need to declare the name or the version. If the recipe declares them, they should match the provided values in the command line.

- The user and channel: user/channel. The command will assume that the name and version are provided by the recipe.

- The version, user and channel: version@user/channel. The recipe must provide the name, and if it does provide the version, it should match the command line one.

The export command will run a linting of the package recipe, looking for possible inconsistencies, bugs and py2-3 incompatibilities. It is possible to customize the rules for this linting, as well as totally disabling it. Look at the recipe_linter and pylintrc variables in conan.conf and the PYLINTRC environment variable.

**Examples**

14.1. Commands

273
• Export a recipe using a full reference. Only valid if name and version are not declared in the recipe:

```
$ conan export . mylib/1.0@myuser/channel
```

• Export a recipe from any folder directory, under the myuser/stable user and channel:

```
$ conan export ./folder_name myuser/stable
```

• Export a recipe without removing the source folder in the local cache:

```
$ conan export . fenix/stable -k
```

### conan export-pkg

```
$ conan export-pkg [-h] [-bf BUILD_FOLDER] [-e ENV] [-f]
     [-if INSTALL_FOLDER] [-o OPTIONS] [-pr PROFILE]
     [-pf PACKAGE_FOLDER] [-s SETTINGS] [-sf SOURCE_FOLDER]
     [-j JSON]
     path reference
```

Exports a recipe, then creates a package from local source and build folders.

If ‘--package-folder’ is provided it will copy the files from there, otherwise it will execute package() method over ‘--source-folder’ and ‘--build-folder’ to create the binary package.

**positional arguments:**

<table>
<thead>
<tr>
<th>path</th>
<th>Path to a folder containing a conanfile.py or to a recipe file e.g., my_folder/conanfile.py</th>
</tr>
</thead>
<tbody>
<tr>
<td>reference</td>
<td>user/channel or pkg/version@user/channel (if name and version are not declared in the conanfile.py)</td>
</tr>
</tbody>
</table>

**optional arguments:**

| -h, --help      | show this help message and exit |
| -bf BUILD_FOLDER, --build-folder BUILD_FOLDER | Directory for the build process. Defaulted to the current directory. A relative path to current directory can also be specified |
| -e ENV, --env ENV | Environment variables that will be set during the package build, e.g., CXX=/usr/bin/clang++ |
| -f, --force     | Overwrite existing package if existing |
| -if INSTALL_FOLDER, --install-folder INSTALL_FOLDER | Directory containing the conaninfo.txt and conanbuildinfo.txt files (from previous 'conan install'). Defaulted to --build-folder. If these files are found in the specified folder and any of '-e', '-o', '-pr' or '-s' arguments are used, it will raise an error. |
| -o OPTIONS, --options OPTIONS | Define options values, e.g., pkg:with_qt=true |
| -pr PROFILE, --profile PROFILE | Profile for this package |
| -pf PACKAGE_FOLDER, --package-folder PACKAGE_FOLDER | Folder containing a locally created package. If a value is given, it won't call the recipe 'package()' method, and will run a copy of the provided folder. |
| -s SETTINGS, --settings SETTINGS |

(continues on next page)
The `export-pkg` command let you create a package from already existing files in your working folder, it can be useful if you are using a build process external to Conan and do not want to provide it with the recipe. Nevertheless, you should take into account that it will generate a package and Conan won’t be able to guarantee its reproducibility or regenerate it again. This is **not** the normal or recommended flow for creating Conan packages.

Execution of this command will result in several files copied to the package folder in the cache identified by its package_id (Conan will perform all the required actions to compute this _id_: build the graph, evaluate the requirements and options, and call any required method), but there could be two different sources for the files:

- If the argument `--package-folder` is provided, Conan will just copy all the contents of that folder to the package one in the cache.
- If no `--package-folder` is given, Conan will execute the method `package()` once and the `self.copy(...)` functions will copy matching files from the source_folder and build_folder to the corresponding path in the Conan cache (working directory corresponds to the build_folder).

There are different scenarios where this command could look like useful:

- **You are working locally on a package** and you want to upload it to the cache to be able to consume it from other recipes. In this situation you can use the `export-pkg` command to copy the package to the cache, but you could also put the **package in editable mode** and avoid this extra step.
- **You only have precompiled binaries available**, then you can use the `export-pkg` to create the Conan package, or you can build a working recipe to download and package them. These scenarios are described in the documentation section **How to package existing binaries**.

**Note:** Note that if `--profile`, settings or options are not provided to `export-pkg`, the configuration will be extracted from the information stored after a previous `conan install`. That information might be incomplete in some edge cases, so we strongly recommend the usage of `--profile` or `--settings, --options` etc.

**Examples**

- **Create a package from a directory containing the binaries for Windows/x86/Release:**

  We need to collect all the files from the local filesystem and tell Conan to compute the proper package_id so its get associated with the correct settings and it works when consuming it.

  If the files in the working folder are:

  ```
  Release_x86/lib/libmycoollib.a
  Release_x86/lib/other.a
  Release_x86/include/mylib.h
  Release_x86/include/other.h
  ```

  then, just run:

  ```
  $ conan new Hello/0.1 --bare  # It creates a minimum recipe example
  $ conan export-pkg . Hello/0.1@user/stable -s os=Windows -s arch=x86 -s build_=type=Release --package-folder=Release_x86
  ```

---

**Table:**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-sf SOURCE_FOLDER, --source-folder SOURCE_FOLDER</code></td>
<td>Directory containing the sources. Defaulted to the conanfile's directory. A relative path to current directory can also be specified</td>
</tr>
<tr>
<td><code>-j JSON, --json JSON</code></td>
<td>Path to a json file where the install information will be written</td>
</tr>
</tbody>
</table>
This last command will copy all the contents from the package-folder and create the package associated with the settings provided through the command line.

- Create a package from a source and build folder:

  The objective is to collect the files that will be part of the package from the source folder (include files) and from the build folder (libraries), so, if these are the files in the working folder:

  ```
  sources/include/mylib.h
  sources/src/file.cpp
  build/lib/mylib.lib
  build/lib/mylib.tmp
  build/file.obj
  ```

  we would need a slightly more complicated `conanfile.py` than in the previous example to select which files to copy, we need to change the patterns in the `package()` method:

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

  Now, we can run Conan to create the package:

  ```
  $ conan export-pkg . Hello/0.1@user/stable -pr=myprofile --source-folder=sources -
  $ conan new
  ```

  **conan new**

  ```
  [-cicco] [-gi] [-ciu CI_UPLOAD_URL]
  name
  ```

  Creates a new package recipe template with a `conanfile.py` and optionally, `test_package` testing files.
<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-cilg, --ci-travis-gcc</td>
<td>Generate travis-ci files for linux gcc</td>
</tr>
<tr>
<td>-cilc, --ci-travis-clang</td>
<td>Generate travis-ci files for linux clang</td>
</tr>
<tr>
<td>-cio, --ci-travis-osx</td>
<td>Generate travis-ci files for OSX apple-clang</td>
</tr>
<tr>
<td>-ciw, --ci-appveyor-win</td>
<td>Generate appveyor files for Appveyor Visual Studio</td>
</tr>
<tr>
<td>-ciglg, --ci-gitlab-gcc</td>
<td>Generate GitLab files for linux gcc</td>
</tr>
<tr>
<td>-ciglc, --ci-gitlab-clang</td>
<td>Generate GitLab files for linux clang</td>
</tr>
<tr>
<td>-ciccg, --ci-circleci-gcc</td>
<td>Generate CircleCI files for linux gcc</td>
</tr>
<tr>
<td>-ciccc, --ci-circleci-clang</td>
<td>Generate CircleCI files for linux clang</td>
</tr>
<tr>
<td>-cicco, --ci-circleci-osx</td>
<td>Generate CircleCI files for OSX apple-clang</td>
</tr>
<tr>
<td>-gi, --gitignore</td>
<td>Generate a .gitignore with the known patterns to excluded</td>
</tr>
<tr>
<td>-ciu CI_UPLOAD_URL, --ci-upload-url CI_UPLOAD_URL</td>
<td>Define URL of the repository to upload</td>
</tr>
</tbody>
</table>

Examples:

- Create a new `conanfile.py` for a new package `mypackage/1.0@myuser/stable`:

  ```bash
  $ conan new mypackage/1.0
  
  $ conan new mypackage/1.0 -t
  
  $ conan new mypackage/1.0@myuser/stable -t -cilg -cio -ciw
  
  $ conan new mypackage/1.0@myuser/stable -t -ciglg -ciglc -ciu https://api.bintray.com/conan/myuser/myrepo
  
  $ conan new mypackage/1.0 --template=myconanfile.py
  
  Conan will look for `templates/myconanfile.py` in the Conan local cache. If an absolute path is given as argument, it will be used instead. These files can be installed and managed by `conan config install` command. The templates use Jinja syntax:

  ```python
  class {{package_name}}Conan(ConanFile):
      name = "{{name}}"
      version = "{{version}}"
  ```

  Where `name` and `version` placeholders are defined from the command line, and `package_name` is a camel case variant of the package name.

14.1. Commands 277
conan Documentation, Release 1.16.0

conan upload

```
$ conan upload [-h] [-p PACKAGE] [-q QUERY] [-r REMOTE] [--all]
              [--retry-wait RETRY_WAIT] [-no [{all,recipe}]] [-j JSON]

pattern_or_reference
```

Uploads a recipe and binary packages to a remote.

If no remote is specified, the first configured remote (by default conan-center, use ‘conan remote list’ to list the remotes) will be used.

**Positional Arguments:**
- `pattern_or_reference`: Pattern, recipe reference or package reference e.g., 'boost/*', 'MyPackage/1.2@user/channel', 'MyPackage/1.2@user/channel:af7901d0bdfde621d086181aa1c495c25a17b137'

**Optional Arguments:**
- `-h, --help`: show this help message and exit
- `-p PACKAGE, --package PACKAGE`: Package ID [DEPRECATED: use full reference instead]
- `-q QUERY, --query QUERY`: Only upload packages matching a specific query.
  Packages query: 'os=Windows AND (arch=x86 OR compiler=gcc)'. The 'pattern_or_reference' parameter has to be a reference: MyPackage/1.2@user/channel
- `-r REMOTE, --remote REMOTE`: upload to this specific remote
- `--all`: Upload both package recipe and packages
- `--skip-upload`: Do not upload anything, just run the checks and the compression
- `--force`: Do not check conan recipe date, override remote with local
- `--check`: Perform an integrity check, using the manifests, before upload
- `--c, --confirm`: Upload all matching recipes without confirmation
- `--retry RETRY`: In case of fail retries to upload again the specified times.
- `--retry-wait RETRY_WAIT`: Waits specified seconds before retry again
- `--no [{all,recipe}]], --no-overwrite [{all,recipe}]`: Uploads package only if recipe is the same as the remote one
- `-j JSON, --json JSON`: json file path where the upload information will be written to

**Examples:**

Uploads a package recipe (`conanfile.py` and the exported files):

```
$ conan upload OpenCV/1.4.0@lasote/stable
```

Uploads a package recipe and a single binary package:

```
$ conan upload OpenCV/1.4.0@lasote/stable:d50a0d523d98c15bb147b18fa7d203887c38be8b
```

Uploads a package recipe and all the generated binary packages to a specified remote:
$ conan upload OpenCV/1.4.0@lasote/stable --all -r my_remote

Uploads all recipes and binary packages from our local cache to my_remote without confirmation:

$ conan upload "*" --all -r my_remote -c

Uploads the recipe for OpenCV alongside any of its binary packages which are built with settings arch=x86_64 and os=Linux from our local cache to my_remote:

$ conan upload OpenCV/1.4.0@lasote/stable -q 'arch=x86_64 and os=Linux' -r my_remote

Upload all local packages and recipes beginning with “Op” retrying 3 times and waiting 10 seconds between upload attempts:

$ conan upload "Op*" --all -r my_remote -c --retry 3 --retry-wait 10

Upload packages without overwriting the recipe and packages if the recipe has changed:

$ conan upload OpenCV/1.4.0@lasote/stable --all --no-overwrite # defaults to --no-overwrite all

Upload packages without overwriting the recipe if the packages have changed:

$ conan upload OpenCV/1.4.0@lasote/stable --all --no-overwrite recipe

### conan test


Tests a package consuming it from a conanfile.py with a test() method.

This command installs the conanfile dependencies (including the tested package), calls a ‘conan build’ to build test apps and finally executes the test() method. The testing recipe does not require name or version, neither definition of package() or package_info() methods. The package to be tested must exist in the local cache or in any configured remote.

<table>
<thead>
<tr>
<th>positional arguments:</th>
</tr>
</thead>
<tbody>
<tr>
<td>path</td>
</tr>
<tr>
<td>reference</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>optional arguments:</th>
</tr>
</thead>
<tbody>
<tr>
<td>-h, --help</td>
</tr>
<tr>
<td>-tbf TEST_BUILD_FOLDER, --test-build-folder TEST_BUILD_FOLDER</td>
</tr>
<tr>
<td>-b [BUILD], --build [BUILD]</td>
</tr>
</tbody>
</table>

(continues on next page)
the nodes with some dependency being built (for any reason). Can be used together with any other build policy. Useful to make sure that any new change introduced in a dependency is incorporated by building again the package. --build=oudated Build from code if the binary is not built with the current recipe or when missing binary package. --build=[pattern] Build always these packages from source, but never build the others. Allows multiple --build parameters. 'pattern' is a fnmatch file pattern of a package reference. Default behavior: If you don't specify anything, it will be similar to '--build=never', but package recipes can override it with their 'build_policy' attribute in the conanfile.py.

- `e ENV, --env ENV` Environment variables that will be set during the package build, `e CXX=/usr/bin/clang++`
- `o OPTIONS, --options OPTIONS` Define options values, e.g., `-o Pkg:with_qt=true`
- `pr PROFILE, --profile PROFILE` Apply the specified profile to the install command
- `r REMOTE, --remote REMOTE` Look in the specified remote server
- `s SETTINGS, --settings SETTINGS` Settings to build the package, overwriting the defaults. e.g., `-s compiler=gcc`
- `u, --update` Check updates exist from upstream remotes

This command is util for testing existing packages, that have been previously built (with `conan create`, for example). `conan create` will automatically run this test if a `test_package` folder is found besides the `conanfile.py`, or if the `--test-folder` argument is provided to `conan create`.

**Example:**

```
$ conan new Hello/0.1 -s -t
$ mv test_package test_package2
$ conan create . user/testing
# doesn't automatically run test, it has been renamed
# now run test
$ conan test test_package2 Hello/0.1@user/testing
```

The test package folder, could be elsewhere, or could be even applied to different versions of the package.

### 14.1.3 Package development commands

Commands related to the local (user space) development of a Conan package:

**conan source**

```
$ conan source [-h] [-sf SOURCE_FOLDER] [-if INSTALL_FOLDER] path
```

Calls your local conanfile.py ‘source()’ method.

Usually downloads and uncompresses the package sources.
positional arguments:
path Path to a folder containing a conanfile.py or to a recipe file e.g., my_folder/conanfile.py

optional arguments:
-h, --help show this help message and exit
-sf SOURCE_FOLDER, --source-folder SOURCE_FOLDER Destination directory. Defaulted to current directory
-if INSTALL_FOLDER, --install-folder INSTALL_FOLDER Directory containing the conaninfo.txt and conanbuildinfo.txt files (from previous 'conan install'). Defaulted to --build-folder Optional, source method will run without the information retrieved from the conaninfo.txt and conanbuildinfo.txt, only required when using conditional source() based on settings, options, env_info and user_info

The source() method might use (optional) settings, options and environment variables from the specified profile and dependencies information from the declared deps_XXX_info objects in the conanfile requirements.

All that information is saved automatically in the conaninfo.txt and conanbuildinfo.txt files respectively, when you run the conan install command. Those files have to be located in the specified --install-folder.

Examples:
• Call a local recipe’s source method: In user space, the command will execute a local conanfile.py source() method, in the src folder in the current directory.

  $ conan new lib/1.0@conan/stable
  $ conan source . --source-folder mysrc

• In case you need the settings/options or any info from the requirements, perform first an install:

  $ conan install . --install-folder mybuild
  $ conan source . --source-folder mysrc --install-folder mybuild

conan build

$ conan build [-h] [-b] [-bf BUILD_FOLDER] [-c] [-i] [-t]
  [-if INSTALL_FOLDER] [-pf PACKAGE_FOLDER]
  [-sf SOURCE_FOLDER]
  path

Calls your local conanfile.py ‘build()’ method.

The recipe will be built in the local directory specified by --build-folder, reading the sources from --source-folder. If you are using a build helper, like CMake(), the --package-folder will be configured as destination folder for the install step.
-b, --build
  Execute the build step (variable should_build=True).
  When specified, configure/install/test won't run
  unless --configure/--install/--test specified

-bf BUILD_FOLDER, --build-folder BUILD_FOLDER
  Directory for the build process. Defaulted to the
  current directory. A relative path to current
  directory can also be specified

-c, --configure
  Execute the configuration step (variable
  should_configure=True). When specified,
  build/install/test won't run unless
  --build/--install/--test specified

-i, --install
  Execute the install step (variable
  should_install=True). When specified,
  configure/build/test won't run unless
  --configure/--build/--test specified

-t, --test
  Execute the test step (variable should_test=True).
  When specified, configure/build/install won't run
  unless --configure/--build/--install specified

-if INSTALL_FOLDER, --install-folder INSTALL_FOLDER
  Directory containing the conaninfo.txt and
  conanbuildinfo.txt files (from previous 'conan
  install'). Defaulted to --build-folder

-pf PACKAGE_FOLDER, --package-folder PACKAGE_FOLDER
  Directory to install the package (when the build
  system or build() method does it). Defaulted to the
  '{build_folder}/package' folder. A relative path can
  be specified, relative to the current folder. Also an
  absolute path is allowed.

-sf SOURCE_FOLDER, --source-folder SOURCE_FOLDER
  Directory containing the sources. Defaulted to the
  conanfile's directory. A relative path to current
  directory can also be specified

The `build()` method might use settings, options and environment variables from the specified profile and dependen-
cies information from the declared `deps_XXX_info` objects in the conanfile requirements. All that information is
saved automatically in the `conaninfo.txt` and `conanbuildinfo.txt` files respectively, when you run the `conan install`
command. Those files have to be located in the specified `--build-folder` or in the `--install-folder` if
specified.

The `--configure, --build, --install` arguments control which parts of the `build()` are actu-
ally executed. They have related conanfile boolean variables `should_configure, should_build, should_install`, which are `True` by default, but that will change if some of these arguments are used in the
command line. The CMake and Meson and AutotoolsBuildEnvironment helpers already use these variables.

**Example**: Building a conan package (for architecture x86) in a local directory.

**Listing 1: conanfile.py**

```python
from conans import ConanFile, CMake, tools

class LibConan(ConanFile):
    ...
    def source(self):
        self.run("git clone https://github.com/conan-io/hello.git")
    def build(self):
```
cmake = CMake(self)
cmake.configure(source_folder="hello")
cmake.build()

First we will call **conan source** to get our source code in the *src* directory, then **conan install** to install the requirements and generate the info files, and finally **conan build** to build the package:

```
$ conan source . --source-folder src
$ conan install . --install-folder build_x86 -s arch=x86
$ conan build . --build-folder build_x86 --source-folder src
```

Or if we want to create the **conaninfo.txt** and **conanbuildinfo.txt** files in a different folder:

```
$ conan source . --source-folder src
$ conan install . --install-folder install_x86 -s arch=x86
$ conan build . --build-folder build_x86 --install-folder install_x86 --source-folder src
```

However, we recommend the **conaninfo.txt** and **conanbuildinfo.txt** to be generated in the same –build-folder, otherwise, you will need to specify a different folder in your build system to include the files generators file. E.g., **conanbuildinfo.cmake**

**Example:** Control the build stages

You can control the build stages using **--configure**/**--build**/**--install**/**--test** arguments. Here is an example using the CMake build helper:

```
$ conan build . --configure       # only run cmake.configure(). Other methods will do nothing
$ conan build . --build           # only run cmake.build(). Other methods will do nothing
$ conan build . --install         # only run cmake.install(). Other methods will do nothing
$ conan build . --test            # only run cmake.test(). Other methods will do nothing
# They can be combined
$ conan build . -c -b             # run cmake.configure() + cmake.build(), but not cmake. --install() nor cmake.test
```

If nothing is specified, all the methods will be called.

**See also:**

Read more about **should_configure**, **should_build**, **should_install**, **should_test**.

**conan package**

```
```

Calls your local conanfile.py ‘package()’ method.

This command works in the user space and it will copy artifacts from the –build-folder and –source-folder folder to the –package-folder one. It won’t create a new package in the local cache, if you want to do it, use ‘conan create’ or ‘conan export-pkg’ after a ‘conan build’ command.
recipe file e.g., my_folder/conanfile.py

optional arguments:
- h, --help  show this help message and exit
- bf BUILD_FOLDER, --build-folder BUILD_FOLDER
  Directory for the build process. Defaulted to the current directory. A relative path to current directory can also be specified.
- if INSTALL_FOLDER, --install-folder INSTALL_FOLDER
  Directory containing the conaninfo.txt and conanbuildinfo.txt files (from previous 'conan install'). Defaulted to --build-folder.
- pf PACKAGE_FOLDER, --package-folder PACKAGE_FOLDER
  Folder to install the package. Defaulted to the '{build_folder}/package' folder. A relative path can be specified (relative to the current directory). Also an absolute path is allowed.
- sf SOURCE_FOLDER, --source-folder SOURCE_FOLDER
  Directory containing the sources. Defaulted to the conanfile's directory. A relative path to current directory can also be specified.

The `package()` method might use *settings*, *options* and *environment variables* from the specified profile and dependencies information from the declared `deps_XXX_info` objects in the conanfile requirements.

All that information is saved automatically in the `conaninfo.txt` and `conanbuildinfo.txt` files respectively, when you run `conan install`. Those files have to be located in the specified `--build-folder`.

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

**Examples**

This example shows how `package()` works in a package which can be edited and built in user folders instead of the local cache.

```
$ conan new Hello/0.1 -s
$ conan install . --install-folder=build_x86 -s arch=x86
$ conan build . --build-folder=build_x86
$ conan package . --build-folder=build_x86 --package-folder=package_x86
$ ls package/x86
> conaninfo.txt conanmanifest.txt include/ lib/
```

**Note:** The packages created locally are just for the user, but cannot be directly consumed by other packages, nor they can be uploaded to a remote repository. In order to make these packages available to the system, they have to be put in the conan local cache, which can be done with the `conan export-pkg` command instead of using `conan package` command:

```
$ conan new Hello/0.1 -s
$ conan install . --install-folder=build_x86 -s arch=x86
$ conan build . --build-folder=build_x86
$ conan export-pkg . Hello/0.1@user/stable --build-folder=build_x86 -s arch=x86
```
conan editable

$ conan editable [-h] {add,remove,list} ...

Manages editable packages (package that resides in the user workspace, but are consumed as if they were in the cache).
Use the subcommands ‘add’, ‘remove’ and ‘list’ to create, remove an list packages currently installed in this mode.

positional arguments:
{add,remove,list} sub-command help
add Put a package in editable mode
remove Disable editable mode for a package
list List packages in editable mode

optional arguments:
-h, --help show this help message and exit

conan editable add

$ conan editable add [-h] [-l LAYOUT] path reference

Opens the package <reference> in editable mode in the user folder <path>

positional arguments:
path Path to the package folder in the user workspace
reference Package reference e.g.: mylib/1.X@user/channel

optional arguments:
-h, --help show this help message and exit
-l LAYOUT, --layout LAYOUT Relative or absolute path to a file containing the layout. Relative paths will be resolved first relative to current dir, then to local cache "layouts" folder

This command puts a package in “Editable mode”, and consumers of this package will use it from the given user folder instead of using it from the cache. The path pointed by path should exist and contain a conanfile.py.

Example: Put the package cool/version@user/dev in editable mode, using the layout specified by the file win_layout.

$ conan editable add . cool/version@user/dev --layout=win_layout

conan editable remove

$ conan editable remove [-h] reference

Removes the editable mode of package reference.

positional arguments:
reference Package reference e.g.: mylib/1.X@user/channel

optional arguments:
-h, --help show this help message and exit
Example: remove the “Editable mode”, use again package from the cache:

```
$ conan editable remove cool/version@user/dev
```

### conan editable list

```
$ conan editable list [-h]
```

Shows the list of the packages that are opened in “editable” mode.

### conan workspace

```
$ conan workspace [-h] {install} ...
```

Manages a workspace (a set of packages consumed from the user workspace that belongs to the same project).

Use this command to manage a Conan workspace, use the subcommand ‘install’ to create the workspace from a file.

Positional arguments:

- **{install}** sub-command help
  - **install** same as a "conan install" command but using the workspace data from the file. If no file is provided, it will look for a file named "conanws.yml"

Optional arguments:

- **-h, --help** show this help message and exit

### conan workspace install

```
$ conan workspace install [-h] [-b [BUILD]] [-e ENV] [-o OPTIONS]
  [-p PR] [-r REMOTE] [-s SETTINGS] [-u]
  [-i INSTALLED_FOLDERS]
  path
```

Positional arguments:

- **path** path to workspace definition file (it will look for a "conanws.yml" inside if a directory is given)

Optional arguments:

- **-h, --help** show this help message and exit

- **-b [BUILD], --build [BUILD]** Optional, use it to choose if you want to build from sources: --build Build all from sources, do not use binary packages. --build=never Never build, use binary packages or fail if a binary package is not found. --build=missing Build from code if a binary package is not found. --build=outdated Build from code if the binary is not built with the current recipe or when missing binary package. --build=[pattern] Build always these packages from source, but never build the others. Allows multiple --build parameters. 'pattern' is a fnmatch file pattern of a package name. Default
behavior: If you don't specify anything, it will be similar to '--build=never', but package recipes can override it with their 'build_policy' attribute in the conanfile.py.

-e ENV, --env ENV Environment variables that will be set during the package build, -e CXX=/usr/bin/clang++

-o OPTIONS, --options OPTIONS Define options values, e.g., -o Pkg:with_qt=true

-pp PROFILE, --profile PROFILE Apply the specified profile to the install command

-r REMOTE, --remote REMOTE Look in the specified remote server

-s SETTINGS, --settings SETTINGS Settings to build the package, overwriting the defaults. e.g., -s compiler=gcc

-u, --update Check updates exist from upstream remotes

-ff INSTALL_FOLDER, --install-folder INSTALL_FOLDER Folder where the workspace files will be created (default to current working directory)

Note that these arguments, like settings and options mostly apply to the dependencies, but those packages that are defined as editable in the workspace are in the user space. Those packages won’t be built by the command (even with --build arguments), as they are built locally. It is the responsibility of the editables layout to match the settings (typically parameterizing the layout with settings and options)

14.1.4 Misc commands

Other useful commands:

conan profile

$ conan profile [-h] {list,show,new,update,get,remove} ...

Lists profiles in the ‘.conan/profiles’ folder, or shows profile details.

The ‘list’ subcommand will always use the default user ‘.conan/profiles’ folder. But the ‘show’ subcommand is able to resolve absolute and relative paths, as well as to map names to ‘.conan/profiles’ folder, in the same way as the ‘--profile’ install argument.

positional arguments:
{list,show,new,update,get,remove}

list List current profiles
show Show the values defined for a profile
new Creates a new empty profile
update Update a profile with desired value
get Get a profile key
remove Remove a profile key

optional arguments:
-h, --help show this help message and exit

Examples

• List the profiles:
$ conan profile list
  > myprofile1
  > myprofile2

• Print profile contents:

$ conan profile show myprofile1
Profile myprofile1
[settings]
...

• Print profile contents (in the standard directory .conan/profiles):

$ conan profile show myprofile1
Profile myprofile1
[settings]
...

• Print profile contents (in a custom directory):

$ conan profile show /path/to/myprofile1
Profile myprofile1
[settings]
...

• Update a setting from a profile located in a custom directory:

$ conan profile update settings.build_type=Debug /path/to/my/profile

• Add a new option to the default profile:

$ conan profile update options.zlib:shared=True default

• Create a new empty profile:

$ conan profile new /path/to/new/profile

• Create a new profile detecting the settings:

$ conan profile new /path/to/new/profile --detect

• Create a new or overwrite an existing profile with detected settings:

$ conan profile new /path/to/new/profile --detect --force

conan remote

$ conan remote [-h]
  (list, add, remove, update, rename, list_ref, add_ref, remove_ref, update_ref,
   list_pref, add_pref, remove_pref, update_pref, clean)
...

Manages the remote list and the package recipes associated to a remote.
### positional arguments:

- `list`, `add`, `remove`, `update`, `rename`, `list_ref`, `add_ref`, `remove_ref`, `update_ref`, `list_pref`, `add_pref`, `remove_pref`, `update_pref`, `clean`

    | sub-command help | positional arguments: |
    |-----------------|----------------------|
    | list            | List current remotes |
    | add             | Add a remote         |
    | remove          | Remove a remote      |
    | update          | Update the remote url|
    | rename          | Update the remote name|
    | list_ref        | List the package recipes and its associated remotes |
    | add_ref         | Associate a recipe’s reference to a remote |
    | remove_ref      | Dissociate a recipe’s reference and its remote |
    | update_ref      | Update the remote associated with a package recipe |
    | list_pref       | List the package binaries and its associated remotes |
    | add_pref        | Associate a package reference to a remote |
    | remove_pref     | Dissociate a package’s reference and its remote |
    | update_pref     | Update the remote associated with a binary package |
    | clean           | Clean the list of remotes and all recipe-remote associations |

### optional arguments:

- `-h`, `--help` show this help message and exit

### Examples

- **List remotes:**

  ```bash
  $ conan remote list
  conan-center: https://conan.bintray.com [Verify SSL: True]
  local: http://localhost:9300 [Verify SSL: True]
  ```

- **List remotes in a format valid for remotes.txt (conan config install):**

  ```bash
  $ conan remote list --raw
  conan-center https://conan.bintray.com True
  local http://localhost:9300 True
  # capture the current remotes in a text file
  $ conan remote list --raw > remotes.txt
  ```

- **Add a new remote:****

  ```bash
  $ conan remote add remote_name remote_url [verify_ssl]
  ```

  Verify SSL option can be True or False (default True). Conan client will verify the SSL certificates.

- **Insert a new remote:**

  Insert as the first one (position/index 0), so it is the first one to be checked:

  ```bash
  $ conan remote add remote_name remote_url [verify_ssl] --insert
  ```

  Insert as the second one (position/index 1), so it is the second one to be checked:

  ```bash
  $ conan remote add remote_name remote_url [verify_ssl] --insert=1
  ```

- **Add or insert a remote:**

  Adding the `--force` argument to `conan remote add` will always work, and won’t raise an error. If an existing remote exists with that remote name or URL, it will be updated with the new information. The `--insert` works the
same. If not specified, the remote will be appended the last one. If specified, the command will insert the remote in the specified position

```bash
$ conan remote add remote_name remote_url [verify_ssl] --force --insert=1
```

- Remove a remote:
  ```bash
  $ conan remote remove remote_name
  ```

- Remove all configured remotes (this will also remove all recipe-remote associations):
  ```bash
  $ conan remote clean
  ```

- Update a remote:
  ```bash
  $ conan remote update remote_name new_url [verify_ssl]
  ```

- Rename a remote:
  ```bash
  $ conan remote rename remote_name new_remote_name
  ```

- Change an existing remote to the first position:
  ```bash
  $ conan remote update remote_name same_url --insert 0
  ```

- List the package recipes and its associated remotes:
  ```bash
  $ conan remote list_ref
  bzip2/1.0.6@lasote/stable: conan.io
  Boost/1.60.0@lasote/stable: conan.io
  zlib/1.2.8@lasote/stable: conan.io
  ```

- Associate a recipe’s reference to a remote:
  ```bash
  $ conan remote add_ref OpenSSL/1.0.2i@conan/stable conan-center
  ```

- Update the remote associated with a package recipe:
  ```bash
  $ conan remote update_ref OpenSSL/1.0.2i@conan/stable local-remote
  ```

**Note:** Check the section *How to manage SSL (TLS) certificates* section to know more about server certificates verification and client certifications management.

---

**conan user**

```bash
$ conan user [-h] [-c] [-p [PASSWORD]] [-r REMOTE] [-j JSON] [name]
```

Authenticates against a remote with user/pass, caching the auth token.

Useful to avoid the user and password being requested later. e.g. while you’re uploading a package. You can have one user for each remote. Changing the user, or introducing the password is only necessary to perform changes in remote packages.
position arguments:
name Username you want to use. If no name is provided it will show the current user

optional arguments:
-h, --help show this help message and exit
-c, --clean Remove user and tokens for all remotes
-p [PASSWORD], --password [PASSWORD]
User password. Use double quotes if password with spacing, and escape quotes if existing. If empty, the password is requested interactively (not exposed)
-r REMOTE, --remote REMOTE Use the specified remote server
-j JSON, --json JSON json file path where the user list will be written to

Examples:

- List my user for each remote:

```
$ conan user
Current user of remote 'conan-center' set to: 'danimtb' [Authenticated]
Current user of remote 'bincrafters' set to: 'None' (anonymous)
Current user of remote 'upload_repo' set to: 'danimtb' [Authenticated]
Current user of remote 'conan-community' set to: 'danimtb' [Authenticated]
Current user of remote 'the_remote' set to: 'None' (anonymous)
```

- Change bar remote user to foo:

```
$ conan user foo -r bar
Changed user of remote 'bar' from 'None' (anonymous) to 'foo'
```

- Change bar remote user to foo, authenticating against the remote and storing the user and authentication token locally, so a later upload won’t require entering credentials:

```
$ conan user foo -r bar -p mypassword
```

- Clean all local users and tokens:

```
$ conan user --clean
```

- Change bar remote user to foo, asking user password to authenticate against the remote and storing the user and authentication token locally, so a later upload won’t require entering credentials:

```
$ conan user foo -r bar -p
Please enter a password for "foo" account:
Change 'bar' user from None (anonymous) to foo
```

Note: The password is not stored in the client computer at any moment. Conan uses JWT, so it gets a token (expirable by the server) checking the password against the remote credentials. If the password is correct, an authentication token will be obtained, and that token is the information cached locally. For any subsequent interaction with the remotes, the Conan client will only use that JWT token.
**conan imports**

```bash
$ conan imports [-h] [-if INSTALL_FOLDER] [-imf IMPORT_FOLDER] [-u] path
```

Calls your local conanfile.py or conanfile.txt ‘imports’ method.

It requires to have been previously installed and have a conanbuildinfo.txt generated file in the --install-folder (defaulted to current directory).

**Positional Arguments:**

- `path` Path to a folder containing a conanfile.py or to a recipe file e.g., my_folder/conanfile.py With --undo option, this parameter is the folder containing the conan_imports_manifest.txt file generated in a previous execution. e.g.: conan imports ./imported_files --undo

**Optional Arguments:**

- `-h, --help` show this help message and exit
- `-if INSTALL_FOLDER, --install-folder INSTALL_FOLDER` Directory containing the conaninfo.txt and conanbuildinfo.txt files (from previous 'conan install'). Defaulted to --build-folder
- `-imf IMPORT_FOLDER, --import-folder IMPORT_FOLDER` Directory to copy the artifacts to. By default it will be the current directory
- `-u, --undo` Undo imports. Remove imported files

The imports() method might use settings, options and environment variables from the specified profile and dependencies information from the declared deps_XXX_info objects in the conanfile requirements.

All that information is saved automatically in the conaninfo.txt and conanbuildinfo.txt files respectively, when you run `conan install`. Those files have to be located in the specified --install-folder.

**Examples**

- Import files from a current conanfile in current directory:
  ```bash
  $ conan install . --no-imports # Creates the conanbuildinfo.txt
  $ conan imports .
  ```

- Remove the copied files (undo the import):
  ```bash
  $ conan imports . --undo
  ```

**conan copy**

```bash
```

Copies conan recipes and packages to another user/channel.

Useful to promote packages (e.g. from “beta” to “stable”) or transfer them from one user to another.

**Positional Arguments:**

- `reference` package reference. e.g., MyPackage/1.2@user/channel
- `user_channel` Destination user/channel. e.g., lasote/testing

(continues on next page)
### optional arguments:

- `-h, --help` show this help message and exit
- `-p PACKAGE, --package PACKAGE` copy specified package ID
- `--all` Copy all packages from the specified package recipe
- `--force` Override destination packages and the package recipe

### Examples

**Promote a package to stable from beta:**

```bash
$ conan copy OpenSSL/1.0.2i@lasote/beta lasote/stable
```

**Change a package’s username:**

```bash
$ conan copy OpenSSL/1.0.2i@lasote/beta foo/beta
```

### conan download

```bash
```

Downloads recipe and binaries to the local cache, without using settings.

It works specifying the recipe reference and package ID to be installed. Not transitive, requirements of the specified reference will NOT be retrieved. Useful together with ‘conan copy’ to automate the promotion of packages to a different user/channel. Only if a reference is specified, it will download all packages from the specified remote. If no remote is specified, it will use the default remote.

### positional arguments:

- `reference pkg/version@user/channel`

### optional arguments:

- `-h, --help` show this help message and exit
- `-p PACKAGE, --package PACKAGE` Force install specified package ID (ignore settings/options)
- `-r REMOTE, --remote REMOTE` look in the specified remote server
- `-re, --recipe` Downloads only the recipe

### Examples

**Download all OpenSSL/1.0.2i@conan/stable binary packages from the remote foo:**

```bash
$ conan download OpenSSL/1.0.2i@conan/stable -r foo
```

**Download a single binary package of OpenSSL/1.0.2i@conan/stable from the remote foo:**

```bash
$ conan download OpenSSL/1.0.2i@conan/stable -r foo -p 8018a4df6e7d2b4630a814fa40c81b85b9182d2
```

**Download only the recipe of package OpenSSL/1.0.2i@conan/stable from the remote foo:**

```bash
$ conan download OpenSSL/1.0.2i@conan/stable -r foo -re
```
**conan remove**

```
$ conan remove [-h] [-b [BUILDS [BUILDS ...]]] [-f] [-l] [-o]
[-p [PACKAGES [PACKAGES ...]]] [-q QUERY] [-r REMOTE] [-s]
[-t] [pattern_or_reference]
```

Removes packages or binaries matching pattern from local cache or remote.

It can also be used to remove temporary source or build folders in the local conan cache. If no remote is specified, the removal will be done by default in the local conan cache.

**positional arguments:**

- `pattern_or_reference` Pattern or package recipe reference, e.g., 'boost/*', 'MyPackage/1.2@user/channel'

**optional arguments:**

- `-h, --help` show this help message and exit
- `-b [BUILDS [BUILDS ...]], --builds [BUILDS [BUILDS ...]]` By default, remove all the build folders or select one, specifying the package ID
- `-f, --force` Remove without requesting a confirmation
- `-l, --locks` Remove locks
- `-o, --outdated` Remove only outdated from recipe packages. This flag can only be used with a reference
- `-p [PACKAGES [PACKAGES ...]], --packages [PACKAGES [PACKAGES ...]]` Select package to remove specifying the package ID
- `-q QUERY, --query QUERY` Packages query: 'os=Windows AND (arch=x86 OR compiler=gcc)'. The 'pattern_or_reference' parameter has to be a reference: MyPackage/1.2@user/channel
- `-r REMOTE, --remote REMOTE` Will remove from the specified remote
- `-s, --src` Remove source folders
- `-t, --system-reqs` Remove system_reqs folders

The `-q` parameter can’t be used along with `-p` nor `-b` parameters.

**Examples:**

- Remove from the local cache the binary packages (the package recipes will not be removed) from all the recipes matching OpenSSL/* pattern:
  
  ```
  $ conan remove OpenSSL/* --packages
  ```

- Remove the temporary build folders from all the recipes matching OpenSSL/* pattern without requesting confirmation:
  
  ```
  $ conan remove OpenSSL/* --builds --force
  ```

- Remove the recipe and the binary packages from a specific remote:
  
  ```
  $ conan remove OpenSSL/1.0.2@lasote/stable -r myremote
  ```

- Remove only Windows OpenSSL packages from local cache:
  
  ```
  $ conan remove OpenSSL/1.0.2@lasote/stable -q "os=Windows"
  ```

- Remove system requirements installation registry for the package name referred globally for all package ids:
$ conan remove --system-reqs package/version@user/channel

This command does not remove the system installed packages, but only the Conan lock to indicate they were installed.

- Remove system requirements installation registry for all packages named package via a wildcard
  
  $ conan remove --system-reqs 'package/*'

- Remove system requirements installation registry for all packages via a wildcard
  
  $ conan remove --system-reqs '*'

**conan alias**

$ conan alias [-h] reference target

Creates and exports an ‘alias package recipe’.

An “alias” package is a symbolic name (reference) for another package (target). When some package depends on an alias, the target one will be retrieved and used instead, so the alias reference, the symbolic name, does not appear in the final dependency graph.

<table>
<thead>
<tr>
<th>positional arguments:</th>
</tr>
</thead>
<tbody>
<tr>
<td>reference</td>
</tr>
<tr>
<td>target</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>optional arguments:</th>
</tr>
</thead>
<tbody>
<tr>
<td>-h, --help</td>
</tr>
</tbody>
</table>

The command:

$ conan alias Hello/0.X@user/testing Hello/0.1@user/testing

Creates and exports a package recipe for Hello/0.X@user/testing with the following content:

```python
from conans import ConanFile

class AliasConanfile(ConanFile):
    alias = "Hello/0.1@user/testing"
```

Such package recipe acts as a “proxy” for the aliased reference. Users depending on Hello/0.X@user/testing will actually use version Hello/0.1@user/testing. The alias package reference will not appear in the dependency graph at all. It is useful to define symbolic names, or behaviors like “always depend on the latest minor”, but defined upstream instead of being defined downstream with version-ranges.

The “alias” package should be uploaded to servers in the same way as regular package recipes, in order to enable usage from servers.

**conan inspect**

*Warning:* This is an experimental feature subject to breaking changes in future releases.
$ conan inspect [-h] [-a [ATTRIBUTE]] [-r REMOTE] [-j JSON] [--raw RAW] path_or_reference

Displays conanfile attributes, like name, version and options. Works locally, in local cache and remote.

positional arguments:
  path_or_reference Path to a folder containing a recipe (conanfile.py) or to a recipe file. e.g., ./my_project/conanfile.py. It could also be a reference

optional arguments:
  -h, --help show this help message and exit
  -a [ATTRIBUTE], --attribute [ATTRIBUTE]
    The attribute to be displayed, e.g "name"
  -r REMOTE, --remote REMOTE
    look in the specified remote server
  -j JSON, --json JSON json output file
  --raw RAW Print just the value of the requested attribute

Examples:

$ conan inspect zlib/1.2.11@conan/stable -a=name -a=version -a=options -a default_options -a=author
name: zlib
version: 1.2.11
options
  shared: [True, False]
default_options: shared=False

$ conan inspect zlib/1.2.11@conan/stable -a=license -a=url
license: http://www.zlib.net/zlib_license.html
url: http://github.com/conan-community/conan-zlib

$ conan inspect zlib/1.2.11@conan/stable --raw=settings ('os', 'arch', 'compiler', 'build_type')

If no specific attributes are defined via -a, then, some default attributes will be displayed:

$ conan inspect zlib/1.2.11@conan/stable
name: zlib
version: 1.2.11
url: http://github.com/conan-community/conan-zlib
license: http://www.zlib.net/zlib_license.html
author: None
description: A Massively Spiffy Yet Delicately Unobtrusive Compression Library (Also Free, Not to Mention Unencumbered by Patents)
generators: cmake
exports: None
exports_sources: ['CMakeLists.txt']
short_paths: False
apply_env: True
build_policy: None
topics: None
settings: ('os', 'arch', 'compiler', 'build_type')
options:
  shared: [True, False]
default_options:
    shared: False

**conan help**

```bash
$ conan help [-h] [command]
```

Shows help for a specific command.

Positional arguments:

command  command

Optional arguments:

- `--help`: show this help message and exit

This command is equivalent to the `--help` and `-h` arguments

**Example:**

```bash
$ conan help get
> Gets a file or list a directory of a given reference or package.
```

`# same as`

```bash
$ conan get -h
```

### 14.1.5 Output

JSON documents generated by the commands:

**Install and Create output**

**Warning:** This is an experimental feature subject to breaking changes in future releases.

The `conan install` and `conan create` provide a `--json` parameter to generate a file containing the information of the installation process.

The output JSON contains a two first level keys:

- **error**: `True` if the install completed without error, `False` otherwise.
- **installed**: A list of installed packages. Each element contains:
  - **recipe**: Document representing the downloaded recipe.
    - **remote**: remote URL if the recipe has been downloaded, `null` otherwise.
    - **cache**: `true/false`. Retrieved from cache (not downloaded).
    - **downloaded**: `true/false`. Downloaded from a remote (not in cache).
    - **time**: ISO 8601 string with the time the recipe was downloaded/retrieved.
    - **error**: `true/false`.  

14.1. Commands  297
● **id**: Reference. E.g., “OpenSSL/1.0.2n@conan/stable”
● **name**: name of the packaged library. E.g., “OpenSSL”
● **version**: version of the packaged library. E.g., “1.0.2n”
● **user**: user of the packaged library. E.g., “conan”
● **channel**: channel of the packaged library. E.g., “stable”
● **dependency**: true/false. Is the package being installed/created or a dependency. Same as `develop conanfile attribute`.

- **packages**: List of elements, representing the binary packages downloaded for the recipe. Normally there will be only 1 element in this list, only in special cases with build requires, private dependencies and settings overridden this list could have more than one element.
  ● **remote**: remote URL if the recipe has been downloaded. null otherwise.
  ● **cache**: true/false. Retrieved from cache (not downloaded).
  ● **downloaded**: true/false. Downloaded from a remote (not in cache).
  ● **time**: ISO 8601 string with the time the recipe was downloaded/retrieved.
  ● **error**: true/false.
  ● **id**: Package ID. E.g., “8018a4df6c7d2b4630a814fa40c81b85b9182d2b”
  ● **cpp_info**: dictionary containing the build information defined in the `package_info` method on the recipe.

Example:

```
$ conan install OpenSSL/1.0.2l@conan/stable --json install.json
```

Listing 2: install.json

```
{
  "error":false,
  "installed":[
  {
    "recipe":{
      "id":"OpenSSL/1.0.2l@conan/stable",
      "downloaded":true,
      "exported":false,
      "error":null,
      "remote":"https://api.bintray.com/conan/conan/conan-center",
      "time":"2018-11-29T11:59:53.601813",
      "dependency":true,
      "name":"OpenSSL",
      "version":"1.0.2l",
      "user":"conan",
      "channel":"stable"
    },
    "packages":[
      {
        "id":"606fdb601e335c2001bdf31d478826b644747077",
        "downloaded":true,
        "exported":false,
        "error":null,
        "remote":"https://api.bintray.com/conan/conan/conan-center"
      }
    ]
  }
}
```
"time":"2018-11-29T12:00:03.874284",
"built":false,
"cpp_info":{
   "includedirs":[
      "include"
   ],
   "libdirs":[
      "lib"
   ],
   "resdirs":[
      "res"
   ],
   "bindirs":[
      "bin"
   ],
   "builddirs":[
      ""
   ],
   "libs":[
      "ssleay32",
      "libeay32",
      "crypt32",
      "msi",
      "ws2_32"
   ],
   "rootpath":"C:/Users/user/.conan/data/OpenSSL/1.0.2l/conan/stable/package/606fdb601e335c2001bdf31d478826b644747077",
   "version":"1.0.21",
   "description":"OpenSSL is an open source project that provides a robust, commercial-grade, and full-featured toolkit for the Transport Layer Security (TLS) and Secure Sockets Layer (SSL) protocols",
   "public_deps":[
      "zlib"
   ]
}
},
{...
"...":...
}
}

Note: As this is a marked as experimental, some fields may be removed or added: fields version and description inside cpp_info will eventually be removed and paths may be changed for absolute ones.

Search output

Warning: This is an experimental feature subject to breaking changes in future releases.

The conan search provides a --json parameter to generate a file containing the information of the search process.

14.1. Commands 299
The output JSON contains a two first level keys:

- **error**: True if the upload completed without error, False otherwise.
- **results**: A list of the remotes with the packages found. Each element contains:
  - **remote**: Name of the remote.
  - **items**: List of the items found in that remote. For each item there will always be a recipe and optionally also packages when searching them.
    - **recipe**: Document representing the uploaded recipe.
      - **id**: Reference, e.g., “OpenSSL/1.0.2n@conan/stable”
    - **packages**: List of elements representing the binary packages found for the recipe.
      - **id**: Package ID, e.g., “8018a4df6e7d2b4630a814fa40c81b85b9182d2b”
      - **options**: Dictionary of options of the package.
      - **settings**: Dictionary with settings of the package.
      - **requires**: List of requires of the package.
      - **outdated**: Boolean to show whether package is outdated from recipe or not.

Examples:

- Search references in all remotes: **conan search eigen* --r all**

```json
{
    "error":false,
    "results": [ 
        {
            "remote":"conan-center",
            "items": [ 
                {
                    "recipe":{
                        "id":"eigen/3.3.4@conan/stable"
                    }
                }
            ]
        },
        {
            "remote":"upload_repo",
            "items": [ 
                {
                    "recipe":{
                        "id":"eigen/3.3.4@danimtb/stable"
                    }
                },
                {
                    "recipe":{
                        "id":"eigen/3.3.4@danimtb/testing"
                    }
                }
            ]
        },
        {
            "remote":"conan-community",
            "items": [ 
            ]
        }
    ]
}
```
• Search packages of a reference in a remote: `conan search paho-c/1.2.0@conan/stable -r conan-center --json search.json`

```json
{
  "error":false,
  "results":[
    {
      "remote":"conan-center",
      "items":[
        {
          "recipe":{
            "id":"paho-c/1.2.0@conan/stable"
          },
          "packages":[
            {
              "id":"0000193ac313953e78a4f8e82528100030ca70ee",
              "options":{
                "shared":"False",
                "asynchronous":"False",
                "SSL":"False"
              },
              "settings":{
                "os":"Linux",
                "arch":"x86_64",
                "compiler":"gcc",
                "build_type":"Debug",
                "compiler.version":"4.9"
              },
              "requires":[]
            },
            {
              "id":"014be746b283391f79d11e4e8af3154344b58223",
              "options":{
                "shared":"False",
                "asynchronous":"False",
                "SSL":"False"
              },
              "settings":{
                "os":"Windows",
                "compiler.threads":"posix",
                "compiler.exception":"seh",
                "arch":"x86_64",
                "compiler":"gcc"
              }
            }
          ]
        }
      ]
    }
  ]
}
```
"build_type"="Debug",
"compiler.version":"5"
},
"requires":[
],
"outdated":false
},
{
"id":"0188020dbfd167611b967ad2fa0e30710d23e920",
"options":{
  "shared":"True",
  "asynchronous":"False",
  "SSL":"False"
},
"settings":{
  "os":"Macos",
  "arch":"x86_64",
  "compiler":"apple-clang",
  "build_type":"Debug",
  "compiler.version":"9.1"
},
"requires":[
],
"outdated":false
},
{
"id":"03369b0caf8c0c8d4bb84d536112596bde4652d",
"options":{
  "shared":"True",
  "asynchronous":"False",
  "SSL":"False"
},
"settings":{
  "os":"Linux",
  "arch":"x86",
  "compiler":"gcc",
  "build_type":"Release",
  "compiler.version":"5"
},
"requires":[
],
"outdated":false
}

• Search references in local cache: conan search paho-c* --json search.json

```json
{
  "error":false,
}
```
"results": [  
    {  
      "remote": "None",
      "items": [
        {  
          "recipe": {  
            "id": "paho-c/1.2.0@danimtb/testing"
          }
        }
      ]
    }
  ]
}

• Search packages of a reference in local cache: **conan search paho-c/1.2.0@danimtb/testing --json search.json**

```json

{  
  "error": false,
  "results": [  
    {  
      "remote": "None",
      "items": [  
        {  
          "recipe": {  
            "id": "paho-c/1.2.0@danimtb/testing"
          },  
          "packages": [  
            {  
              "id": "6cc50b139b9c3d27b3e9042d5f5372d327b3a9f7",
              "options": {  
                "SSL": "False",
                "asynchronous": "False",
                "shared": "False"
              },  
              "settings": {  
                "arch": "x86_64",
                "build_type": "Release",
                "compiler": "Visual Studio",
                "compiler.runtime": "MD",
                "compiler.version": "15",
                "os": "Windows"
              },  
              "requires": [  
                ]
            },  
            {  
              "id": "95cd13dfc3f6b80d3cccb2a38441e3a1ad88e5a15",
              "options": {  
                "SSL": "False",
                "asynchronous": "True",
                "shared": "True"
              },  
              "settings": {
            }
          ]
        }
      ]
    }
  ]
}
```

(continues on next page)
"arch":"x86_64",
"build_type":"Release",
"compiler":"Visual Studio",
"compiler.runtime":"MD",
"compiler.version":"15",
"os":"Windows"
},
"requires":[
],
"outdated":true
},
{
"id":"970e773c5651dc2560f86200a4ea56c23f568ff9",
"options":{
"SSL":"False",
"asynchronous":"False",
"shared":"True"
},
"settings":{
"arch":"x86_64",
"build_type":"Release",
"compiler":"Visual Studio",
"compiler.runtime":"MD",
"compiler.version":"15",
"os":"Windows"
},
"requires":[
],
"outdated":true
},
{
"id":"c4c0a49b09575515ce1dd9841a48de0c508b9d7c",
"options":{
"SSL":"True",
"asynchronous":"False",
"shared":"True"
},
"settings":{
"arch":"x86_64",
"build_type":"Release",
"compiler":"Visual Studio",
"compiler.runtime":"MD",
"compiler.version":"15",
"os":"Windows"
},
"requires":[
"OpenSSL/1.0.2n@conan/stable:606f6b601e335c2001bdf31d478826b644747077",
"zlib/1.2.11@conan/stable:6cc50b139b9c3d27b3e9042d5f5372d327b3a9f7"
],
"outdated":true
},
{
"id":"db9d6ba7004592ed2598f2c369484d4a01269110",
(continued on next page)
"options":{
  "SSL":"True",
  "asynchronous":"False",
  "shared":"True"
},
"settings":{
  "arch":"x86_64",
  "build_type":"Release",
  "compiler":"gcc",
  "compiler.exception":"seh",
  "compiler.threads":"posix",
  "compiler.version":"7",
  "os":"Windows"
},
"requires":[
  "OpenSSL/1.0.2n@conan/stable:f761d91ce7988eaf86a6179f4cf261609f26",
  "zlib/1.2.11@conan/stable:6dc82da13f94df549e60f9c1ce4c5d11285a4dff"
],
"outdated":true
}

Upload output

Warning: This is an experimental feature subject to breaking changes in future releases.

The conan upload provides a --json parameter to generate a file containing the information of the upload process.

The output JSON contains a two first level keys:

- **error**: True if the upload completed without error, False otherwise.
- **uploaded**: A list of uploaded packages. Each element contains:
  - **recipe**: Document representing the uploaded recipe.
    - id: Reference, e.g., “OpenSSL/1.0.2n@conan/stable”
    - remote_name: Remote name where the recipe was uploaded.
    - remote_url: Remote URL where the recipe was uploaded.
    - time: ISO 8601 string with the time the recipe was uploaded.
  - **packages**: List of elements, representing the binary packages uploaded for the recipe.
    - id: Package ID, e.g., “8018a4df6e7d2b4630a814fa40c81b85b9182d2b”
    - time: ISO 8601 string with the time the recipe was uploaded.
Example:

```bash
$ conan upload h* -all -r conan-center --json upload.json
```

Listing 3: upload.json

```json
{
  "error":false,
  "uploaded":[
    {
      "recipe":{
        "id":"Hello/0.1@conan/testing",
        "remote_name":"conan-center",
        "remote_url":"https://conan.bintray.com",
        "time":"2018-04-30T11:18:19.204728"
      },
      "packages":[
        {
          "id":"3f3387d49612e03a5306289405a2101383b861f0",
          "remote_name":"conan-center",
          "remote_url":"https://conan.bintray.com",
        },
        {
          "id":"6cc50b139b93d27b3e9042d5f5372d327b3a9f7",
          "remote_name":"conan-center",
          "remote_url":"https://conan.bintray.com",
          "time":"2018-04-30T11:18:23.934152"
        },
        {
          "id":"889d5d7812b472f03e021388d338d5f327d8327b3a9f7",
          "remote_name":"conan-center",
          "remote_url":"https://conan.bintray.com",
          "time":"2018-04-30T11:18:28.195266"
        },
        {
          "id":"e98aac150565fc710dff1d4fbee382b087c3ad1d",
          "remote_name":"conan-center",
          "remote_url":"https://conan.bintray.com",
          "time":"2018-04-30T11:18:30.495989"
        }
      ]
    },
    {
      "recipe":{
        "id":"Hello0/1.2.1@conan/testing",
        "remote_name":"conan-center",
        "remote_url":"https://conan.bintray.com",
        "time":"2018-04-30T11:18:32.688651"
      },
      "packages":[
        {
          "id":"5ab84d6acfe1f23c4fae0ab88f26e3a396351ac9",
          "remote_name":"conan-center",
          "remote_url":"https://conan.bintray.com",
          "time":"2018-04-30T11:18:34.991721"
        }
      ]
    }
  ]
}
```

(continues on next page)
"recipe": {
  "id": "HelloApp/0.1@conan/testing",
  "remote_name": "conan-center",
  "remote_url": "https://conan.bintray.com",
  "time": "2018-04-30T11:18:36.901333"
},
"packages": [
  {
    "id": "6cc50b139b9c3d27b3e9042d5f5372d327b3a9f7",
    "remote_name": "conan-center",
    "remote_url": "https://conan.bintray.com",
    "time": "2018-04-30T11:18:39.243895"
  }
],

"recipe": {
  "id": "HelloPythonConan/0.1@conan/testing",
  "remote_name": "conan-center",
  "remote_url": "https://conan.bintray.com",
  "time": "2018-04-30T11:18:41.181543"
},
"packages": [
  {
    "id": "5ab84d6acfe1f23c4fae0ab88f26e3a3963351ac9",
    "remote_name": "conan-center",
    "remote_url": "https://conan.bintray.com",
    "time": "2018-04-30T11:18:43.749422"
  }
],

"recipe": {
  "id": "HelloPythonReuseConan/0.1@conan/testing",
  "remote_name": "conan-center",
  "remote_url": "https://conan.bintray.com",
  "time": "2018-04-30T11:18:45.614096"
},
"packages": [
  {
    "id": "6a051b2648c89dbd1f8ada0031105b287deea9d2",
    "remote_name": "conan-center",
    "remote_url": "https://conan.bintray.com",
    "time": "2018-04-30T11:18:47.942491"
  }
],

"recipe": {
  "id": "hdf5/1.8.20@acri/testing",
  "remote_name": "conan-center",
  "remote_url": "https://conan.bintray.com",
  "time": "2018-04-30T11:18:48.291756"
},
"packages": [ (continues on next page)
User output

Warning: This is an experimental feature subject to breaking changes in future releases.

The conan user provides a --json parameter to generate a file containing the information of the users configured per remote.

The output JSON contains a two first level keys:

- **error**: Boolean indicating whether command completed with error.
- **remotes**: A list of the remotes with the packages found. Each element contains:
  - **name**: Name of the remote.
  - **user_name**: Name of the user set for that remote.
  - **authenticated**: Boolean indicating if user is authenticated or not.

Example:

List users per remote: conan user --json user.json

Listing 4: user.json

```json
{
  "error":false,
  "remotes":[
    {
      "name": "conan-center",
      "user_name": "danimtb",
      "authenticated": true
    },
    {
      ...
    }
  ]
}
```
Info output

Warning: This is an experimental feature subject to breaking changes in future releases.

The `conan info` provides a `--json` parameter to generate a file containing the output of the command. There are several possible outputs depending on other arguments:

**Build order**

The build order printed with the argument `--build-order` can be formatted as JSON. It will show a list of lists where the references inside each nested one can be built in parallel.

```
Listing 5: build_order.json
{
  "groups": [ 
    [ "LibA/0.1@lasote/stable",
      "LibE/0.1@lasote/stable",
      "LibF/0.1@lasote/stable"
    ],
    [ "LibB/0.1@lasote/stable",
      "LibC/0.1@lasote/stable"
    ]
  ]
}
```

**Nodes to build**

When called with the argument `--build` it will retrieve the list of nodes to be built according to the build policy. Output will be just a list of references.
Listing 6: nodes_to_build.json

```
[ "H0/0.1@lu/st",
  "H1a/0.1@lu/st",
  "H1c/0.1@lu/st",
  "H2a/0.1@lu/st",
  "H2c/0.1@lu/st"
]
```

Info output

The output of a **conan info** call over a reference or a path gives information about all the nodes involved in its build graph; the generated JSON file will contain a list with the information for each of the nodes.

Listing 7: info.json

```
[
  {
    "reference": "LibA/0.1@lasote/stable",
    "is_ref": true,
    "display_name": "LibA/0.1@lasote/stable",
    "id": "8da7d879f40d12efabc9a1f26ab12f1b6cafb6ad",
    "build_id": null,
    "url": "myurl",
    "license": [
      "MIT"
    ],
    "recipe": "No remote",
    "binary": "Missing",
    "creation_date": "2019-01-29 17:22:41",
    "required_by": [
      "LibC/0.1@lasote/stable",
      "LibB/0.1@lasote/stable"
    ]
  },
  {
    "reference": "LibB/0.1@lasote/stable",
    "is_ref": true,
    "display_name": "LibB/0.1@lasote/stable",
    "id": "c4ec2bf350e2a0245029ab366a535e26372a4f63",
    "build_id": null,
    "url": "myurl",
    "license": [
      "MIT"
    ],
    "recipe": "No remote",
    "binary": "Missing",
    "creation_date": "2019-01-29 17:22:41",
    "required_by": [
      "conanfile.py (LibD/0.1@None/None)"
    ],
    "requires": [
      "LibA/0.1@lasote/stable",
      "LibE/0.1@lasote/stable"
    ]
  }
]
```

(continues on next page)
14.1.6 Return codes

Return Codes

The Conan client returns different exit codes for every command depending on the situation:

Success

Return code: 0
Execution terminated successfully

General error

Return code: 1
Execution terminated with a general error, normally caused by a ConanException.

Migration error

Return code: 2
Execution terminated with an error migrating configuration files to new format.

User Ctrl+C

Return code: 3
Execution terminated due to manually stopping the process with Ctrl+C key combination.

User Ctrl+Break

Return code: 4
Execution terminated due to manually stopping the process with Ctrl+Break key combination.
SIGTERM

Return code: 5
Execution terminated due to SIGTERM signal.

Invalid configuration

Return code: 6
Execution terminated due to an exception caused by a ConanInvalidConfiguration. This exit code can be considered a success as it is expected for configurations not supported by the recipe.

14.2 conanfile.txt

Reference for conanfile.txt sections: requires, generators, etc.

14.2.1 Sections

[requires]
List of requirements, specifying the full reference.

```
[requires]
Poco/1.9.0@pocoproject/stable
zlib/1.2.11@conan/stable
```

This section supports references with version ranges:

```
[requires]
Poco/]>1.0,<1.8@pocoproject/stable
zlib/1.2.11@conan/stable
```

[build_requires]
List of build requirements specifying the full reference.

```
[build_requires]
7z_installer/1.0@conan/stable
```

This section supports references with version ranges.

In practice the [buildRequires] will be always installed (same as [requires]) as installing from a conanfile.txt means that something is going to be built, so the build requirements are indeed needed.

It is useful and conceptually cleaner to have them in separate sections, so users of this conanfile.txt might quickly identify some dev-tools that they have already installed on their machine, differentiating them from the required libraries to link with.
[generators]

List of generators.

```
[requires]
Poco/1.9.0@pocoproject/stable
zlib/1.2.11@conan/stable

[generators]
xcode
cmake
qmake
```

[options]

List of options scoped for each package like package_name:option = Value.

```
[requires]
Poco/1.9.0@pocoproject/stable
zlib/1.2.11@conan/stable

[generators]
cmake

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

[imports]

List of files to be imported to a local directory. Read more: imports.

```
[requires]
Poco/1.9.0@pocoproject/stable
zlib/1.2.11@conan/stable

[generators]
cmake

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

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

The first item is the subfolder of the packages (could be the root “.” one), the second is the pattern to match. Both relate to the local cache. The third (after the arrow) item, is the destination folder, living in user space, not in the local cache.

The [imports] section also support the same arguments as the equivalent imports() method in conanfile.py, separated with an @.
conan Documentation, Release 1.16.0

Note: If your previous folders use an @ in the path name, use a trailing (even if empty) @ so the parser correctly gets the folders paths, e.g: `lib, * -> /home/jenkins/workspace/conan_test@2/g/install/lib @`

- **root_package** (Optional, Defaulted to all packages in deps): fnmatch pattern of the package name (“OpenCV”, “Boost”) from which files will be copied.
- **folder**: (Optional, Defaulted to False). If enabled, it will copy the files from the local cache to a subfolder named as the package containing the files. Useful to avoid conflicting imports of files with the same name (e.g. License).
- **ignore_case**: (Optional, Defaulted to False). If enabled will do a case-insensitive pattern matching.
- **excludes**: (Optional, Defaulted to None). Allows defining a list of patterns (even a single pattern) to be excluded from the copy, even if they match the main pattern.
- **keep_path** (Optional, Defaulted to True): Means if you want to keep the relative path when you copy the files from the *src* folder to the *dst* one. Useful to ignore (keep_path=False) path of library.dll files in the package it is imported from.

Example to collect license files from dependencies into a *licenses* folder, excluding (just an example) *.html* and *.jpeg* files:

```python
[imports] ., license* -> ./licenses @ folder=True, ignore_case=True, excludes=*.html *.jpeg
```

**Comments**

A comment starts with a hash character (#) and ends at the end of the physical line. Comments are ignored by the syntax; they are not tokens.

### 14.3 conanfile.py

Reference for *conanfile.py*: attributes, methods, etc.

**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:**

#### 14.3.1 Attributes

**name**

This is a string, with a minimum of 2 and a maximum of 50 characters (though shorter names are recommended), that defines the package name. It will be the `<PkgName>/version@user/channel` of the package reference. It
should match the following regex `^[a-zA-Z0-9_]\[a-zA-Z0-9_\+\.-\]{1,50}$`, so start with alphanumeric or underscore, then alphanumeric, underscore, +, .., - characters.

The name is only necessary for `export`-ing the recipe into the local cache (`export` and `create` commands), if they are not defined in the command line. It might take its value from an environment variable, or even any python code that defines it (e.g. a function that reads an environment variable, or a file from disk). However, the most common and suggested approach would be to define it in plain text as a constant, or provide it as command line arguments.

**version**

The version attribute will define the version part of the package reference: `PkgName/<version>@user/channel`. It is a string, and can take any value, matching the same constraints as the `name` attribute. In case the version follows semantic versioning in the form `X.Y.Z-pre1+build2`, that value might be used for requiring this package through version ranges instead of exact versions.

The version is only strictly necessary for `export`-ing the recipe into the local cache (`export` and `create` commands), if they are not defined in the command line. It might take its value from an environment variable, or even any python code that defines it (e.g. a function that reads an environment variable, or a file from disk). Please note that this value might be used in the recipe in other places (as in `source()` method to retrieve code from elsewhere), making this value not constant means that it may evaluate differently in different contexts (e.g., on different machines or for different users) leading to unrepeatable or unpredictable results. The most common and suggested approach would be to define it in plain text as a constant, or provide it as command line arguments.

**description**

This is an optional, but strongly recommended text field, containing the description of the package, and any information that might be useful for the consumers. The first line might be used as a short description of the package.

```python
class HelloConan(ConanFile):
    name = "Hello"
    version = "0.1"
    description = """This is a Hello World library. A fully featured, portable, C++ library to say Hello World in the stdout, with incredible iostreams performance"""
```

**homepage**

Use this attribute to indicate the home web page of the library being packaged. This is useful to link the recipe to further explanations of the library itself like an overview of its features, documentation, FAQ as well as other related information.

```python
class EigenConan(ConanFile):
    name = "eigen"
    version = "3.3.4"
    homepage = "http://eigen.tuxfamily.org"
```

**url**

It is possible, even typical, if you are packaging a third party lib, that you just develop the packaging code. Such code is also subject to change, often via collaboration, so it should be stored in a VCS like git, and probably put on GitHub or a similar service. If you do indeed maintain such a repository, please indicate it in the `url` attribute, so that it can be easily found.
class HelloConan(ConanFile):
    name = "Hello"
    version = "0.1"
    url = "https://github.com/conan-io/hello.git"

The `url` is the url of the package repository, i.e. not necessarily the original source code. It is optional, but highly recommended, that it points to GitHub, Bitbucket or your preferred code collaboration platform. Of course, if you have the conanfile inside your library source, you can point to it, and afterwards use the `url` in your `source()` method.

This is a recommended, but not mandatory attribute.

**license**

This field is intended for the license of the target source code and binaries, i.e. the code that is being packaged, not the `conanfile.py` itself. This info is used to be displayed by the `conan info` command and possibly other search and report tools.

class HelloConan(ConanFile):
    name = "Hello"
    version = "0.1"
    license = "MIT"

This attribute can contain several, comma separated licenses. It is a text string, so it can contain any text, including hyperlinks to license files elsewhere.

However, we strongly recommend packagers of Open-Source projects to use [SPDX](https://spdx.org/) identifiers from the [SPDX license list](https://spdx.org/licenses/) instead of free-formed text. This will help people wanting to automate license compatibility checks, like consumers of your package, or you if your package has Open-Source dependencies.

This is a recommended, but not mandatory attribute.

**author**

Intended to add information about the author, in case it is different from the Conan user. It is possible that the Conan user is the name of an organization, project, company or group, and many users have permissions over that account. In this case, the author information can explicitly define who is the creator/maintainer of the package.

class HelloConan(ConanFile):
    name = "Hello"
    version = "0.1"
    author = "John J. Smith (john.smith@company.com)"

This is an optional attribute.

**topics**

Topics provide a useful way to group related tags together and to quickly tell developers what a package is about. Topics also make it easier for customers to find your recipe. It could be useful to filter packages by topics or to reuse them in Bintray package page.

The `topics` attribute should be a tuple with the needed topics inside.
class ProtocInstallerConan(ConanFile):
    name = "protoc_installer"
    version = "0.1"
    topics = ("protocol-buffers", "protocol-compiler", "serialization", "rpc")

This is an optional attribute.

user, channel

The fields user and channel can be accessed from within a conanfile.py. Though their usage is usually not encouraged, it could be useful in different cases, e.g. to define requirements with the same user and channel as the current package, which could be achieved with something like:

```python
from conans import ConanFile

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

    def requirements(self):
        self.requires("Say/0.1@%s/%s" % (self.user, self.channel))
```

Only package recipes that are in the conan local cache (i.e. "exported") have a user/channel assigned. For package recipes working in user space, there is no current user/channel by default. They can be defined at conan install time with:

```
$ conan install <path to conanfile.py> user/channel
```

If they are not defined via command line, the properties self.user and self.channel will then look for environment variables CONAN_USERNAME and CONAN_CHANNEL respectively. If they are not defined, an error will be raised unless default_user and default_channel are declared in the recipe.

See also:
FAQ: Is there any recommendation regarding which <user> or <channel> to use in a reference?

default_user, default_channel

For package recipes working in the user space, with local methods like conan install . and conan build ., there is no current user/channel. If you are accessing to self.user or self.channel in your recipe, you need to declare the environment variables CONAN_USERNAME and CONAN_CHANNEL or you can set the attributes default_user and default_channel. You can also use python @properties:

```python
from conans import ConanFile

class HelloConan(ConanFile):
    name = "Hello"
    version = "0.1"
    default_user = "myuser"

    @property
    def default_channel(self):
        return "mydefaultchannel"
```

(continues on next page)
```python
def requirements(self):
    self.requires("Pkg/0.1@%s/%s" % (self.user, self.channel))
```

**settings**

There are several things that can potentially affect a package being created, i.e. the final package will be different (a different binary, for example), if some input is different.

Development project-wide variables, like the compiler, its version, or the OS itself. These variables have to be defined, and they cannot have a default value listed in the conanfile, as it would not make sense.

It is obvious that changing the OS produces a different binary in most cases. Changing the compiler or compiler version changes the binary too, which might have a compatible ABI or not, but the package will be different in any case.

For these reasons, the most common convention among Conan recipes is to distinguish binaries by the following four settings, which is reflected in the `conanfile.py` template used in the `conan new` command:

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

When Conan generates a compiled binary for a package with a given combination of the settings above, it generates a unique ID for that binary by hashing the current values of these settings.

But what happens for example to header only libraries? The final package for such libraries is not binary and, in most cases it will be identical, unless it is automatically generating code. We can indicate that in the conanfile:

```python
from conans import ConanFile

class HelloConan(ConanFile):
    name = "Hello"
    version = "0.1"
    # We can just omit the settings attribute too
    settings = None

    def build(self):
        #empty too, nothing to build in header only
```

You can restrict existing settings and accepted values as well, by redeclaring the settings attribute:

```python
class HelloConan(ConanFile):
    settings = ("os": ["Windows"],
                "compiler": {"Visual Studio": {"version": [11, 12]}},
                "arch": None)
```

In this example we have just defined that this package only works in Windows, with VS 10 and 11. Any attempt to build it in other platforms with other settings will throw an error saying so. We have also defined that the runtime (the MD and MT flags of VS) is irrelevant for us (maybe we using a universal one?). Using None as a value means, *maintain the original values* in order to avoid re-typing them. Then, “arch”: None is totally equivalent to “arch”: ["x86", "x86_64", "arm"] Check the reference or your ~/.conan/settings.yml file.

As re-defining the whole settings attribute can be tedious, it is sometimes much simpler to remove or tune specific fields in the `configure()` method. For example, if our package is runtime independent in VS, we can just remove that setting field:
settings = "os", "compiler", "build_type", "arch"

def configure(self):
    self.settings.compiler["Visual Studio"].remove("runtime")

It is possible to check the settings to implement conditional logic, with attribute syntax:

def build(self):
    if self.settings.os == "Windows" and self.settings.compiler.version == "15":
        # do some special build commands
    elif self.settings.arch == "x86_64":
        # Other different commands

Those comparisons do content checking, for example if you do a typo like self.settings.os == "Windos", conan will fail and tell you that is not a valid settings.os value, and the possible range of values.

Likewise, if you try to access some setting that doesn’t exist, like self.settings.compiler.libcxx for the Visual Studio 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
    arch = self.settings.get_safe("arch")
    # Will be None if doesn’t exist
    compiler_version = self.settings.get_safe("compiler.version")

The get_safe() method will return None if that setting or subsetting doesn’t exist.

options

Conan packages recipes can generate different binary packages when different settings are used, but can also customize, per-package any other configuration that will produce a different binary.

A typical option would be being shared or static for a certain library. Note that this is optional, different packages can have this option, or not (like header-only packages), and different packages can have different values for this option, as opposed to settings, which typically have the same values for all packages being installed (though this can be controlled too, defining different settings for specific packages)

Options are defined in package recipes as dictionaries of name and allowed values:

```python
class MyPkg(ConanFile):
    ...
    options = {"shared": [True, False]}
```

Options are defined as a python dictionary inside the ConanFile where each key must be a string with the identifier of the option and the value be a list with all the possible option values:

```python
class MyPkg(ConanFile):
    ...
    options = {"shared": [True, False],
               "option1": ["value1", "value2"],}
```

Values for each option can be typed or plain strings, and there is a special value, ANY, for options that can take any value.

The attribute default_options has the purpose of defining the default values for the options if the consumer (consuming recipe, project, profile or the user through the command line) does not define them. It is worth noticing...
that an uninitialized option will get the value None and it will be a valid value if its contained in the list of valid values. This attribute should be defined as a python dictionary too, although other definitions could be valid for legacy reasons.

```python
class MyPkg(ConanFile):
    ...
    options = {
        "shared": [True, False],
        "option1": ["value1", "value2"],
        "option2": "ANY"
    }
    default_options = {
        "shared": True,
        "option1": "value1",
        "option2": 42
    }

    def build(self):
        shared = "-DBUILD_SHARED_LIBS=ON" if self.options.shared else ""
        cmake = CMake(self)
        self.run("cmake . %s %s" % (cmake.command_line, shared))
        ...
```

**Tip:**

- You can inspect available package options reading the package recipe, which can be done with the command conan inspect MyPkg/0.1@user/channel.
- Options "shared": [True, False] and "fPIC": [True, False] are automatically managed in CMake and AutoToolsBuildEnvironment (configure/make) build helpers.

As we mentioned before, values for options in a recipe can be defined using different ways, let’s go over all of them for the example recipe `MyPkg` defined above:

- Using the attribute `default_options` in the recipe itself.
- In the `default_options` of a recipe that requires this one: the values defined here will override the default ones in the recipe.

```python
class OtherPkg(ConanFile):
    requires = "MyPkg/0.1@user/channel"
    default_options = {
        "MyPkg:shared": False
    }
```

Of course, this will work in the same way working with a `conanfile.txt`:

```
[requires]
MyPkg/0.1@user/channel

[options]
MyPkg:shared=False
```

- It is also possible to define default values for the options of a recipe using profiles. They will apply whenever that recipe is used:

```
# file "myprofile"
# use it as $ conan install -pr=myprofile
[settings]
setting=value

[options]
MyPkg:shared=False
```
• Last way of defining values for options, with the highest priority over them all, is to pass these values using the command argument `-o` in the command line:

```bash
$ conan install . -o MyPkg:shared=True -o OtherPkg:option=value
```

Values for options can be also conditionally assigned (or even deleted) in the methods `configure()` and `config_options()`, the corresponding section has examples documenting these use cases. However, conditionally assigning values to options can have it drawbacks as it is explained in the mastering section.

One important notice is how these options values are evaluated and how the different conditionals that we can implement in Python will behave. As seen before, values for options can be defined in Python code (assigning a dictionary to `default_options`) or through strings (using a `conanfile.txt`, a profile file, or through the command line).

In order to provide a consistent implementation take into account these considerations:

• Evaluation for the typed value and the string one is the same, so all these inputs would behave the same:
  ```python
  – default_options ={"shared": True, "option": None}
  – default_options ={"shared": "True", "option": "None"}
  – MyPkg:shared=True,MyPkg:option=None on profiles, command line or conanfile.txt
  ```

• **Implicit conversion to boolean is case insensitive**, so the expression `bool(self.options.option):`
  ```python
  – 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.
  ```

• Comparison using `is` is always equals to False because the types would be different as the option value is encapsulated inside a Conan class.

• Explicit comparisons with the `==` symbol are case sensitive, so:
  ```python
  – self.options.option = "False" satisfies assert self.options.option == False, assert self.options.option == "False", but assert self.options.option != "false".
  ```

• A different behavior has self.options.option = None, because assert self.options.option != None.

### default_options

As you have seen in the examples above, recipe’s default options are declared as a dictionary with the initial desired value of the options. However, you can also specify default option values of the required dependencies:

```python
class OtherPkg(ConanFile):
    requires = "Pkg/0.1@user/channel"
    default_options = {"Pkg:pkg_option": "value"}
```

And it also works with default option values of conditional required dependencies:

```python
class OtherPkg(ConanFile):
    default_options = {"Pkg:pkg_option": "value"}

    def requirements(self):
        if self.settings.os != "Windows":
            self.requires("Pkg/0.1@user/channel")
```

14.3. conanfile.py
For this example running in Windows, the default_options for the Pkg/0.1@user/channel will be ignored, they will only be used on every other OS.

You can also set the options conditionally to a final value with config_options() instead of using default_options:

```python
class OtherPkg(ConanFile):
    settings = "os", "arch", "compiler", "build_type"
    options = {"some_option": [True, False]}
    # Do NOT declare 'default_options', use 'config_options()'

    def config_options(self):
        if self.options.some_option == None:
            if self.settings.os == 'Android':
                self.options.some_option = True
            else:
                self.options.some_option = False
```

**Important:** Setting options conditionally without a default value works only to define a default value if not defined in command line. However, doing it this way will assign a final value to the option and not an initial one, so those option values will not be overridable from downstream dependent packages.

**Important:** Default options can be specified as a dictionary only for Conan version >= 1.8.

See also:
Read more about the config_options() method.

**requires**
Specify package dependencies as a list of other packages:

```python
class MyLibConan(ConanFile):
    requires = "Hello/1.0@user/stable", "OtherLib/2.1@otheruser/testing"
```

You can specify further information about the package requirements:

```python
class MyLibConan(ConanFile):
    requires = (("Hello/0.1@user/testing"),
                ("Say/0.2@dummy/stable", "override"),
                ("Bye/2.1@coder/beta", "private"))
```

Requirements can be complemented by 2 different parameters:

**private:** a dependency can be declared as private if it is going to be fully embedded and hidden from consumers of the package. Typical examples could be a header only library which is not exposed through the public interface of the package, or the linking of a static library inside a dynamic one, in which the functionality or the objects of the linked static library are not exposed through the public interface of the dynamic library.

**override:** packages can define overrides of their dependencies, if they require the definition of specific versions of the upstream required libraries, but not necessarily direct dependencies. For example, a package can depend on A(v1.0), which in turn could conditionally depend on Zlib(v2), depending on whether the compression is enabled or not. Now, if you want to force the usage of Zlib(v3) you can:
class HelloConan(ConanFile):
    requires = ("A/1.0@user/stable", ("Zlib/3.0@other/beta", "override"))

This **will not introduce a new dependency**, it will just change Zlib v2 to v3 if A actually requires it. Otherwise Zlib will not be a dependency of your package.

**Note:** To prevent accidental override of transitive dependencies, check the config variable `general.error_on_override` or the environment variable `CONAN_ERROR_ON_OVERRIDE`.

**version ranges**

The syntax is using brackets:

```python
class HelloConan(ConanFile):
    requires = "Pkg/[^1.0 <1.8]@user/stable"
```

Expressions are those defined and implemented by [python node-semver](https://pypi.org/project/node-semver/). Accepted expressions would be:

- `>1.1 <2.1`  
  # In such range
- `2.8`  
  # equivalent to =2.8
- `~=3.0`  
  # compatible, according to semver
- `>1.1 || 0.8`  
  # conditions can be OR'ed

Go to [Mastering/Version Ranges](#) if you want to learn more about version ranges.

**build_requires**

Build requirements are requirements that are only installed and used when the package is built from sources. If there is an existing pre-compiled binary, then the build requirements for this package will not be retrieved.

They can be specified as a comma separated tuple in the package recipe:

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

Read more: [Build requirements](#)

**exports**

If a package recipe `conanfile.py` requires other external files, like other python files that it is importing (python importing), or maybe some text file with data it is reading, those files must be exported with the `exports` field, so they are stored together, side by side with the `conanfile.py` recipe.

The `exports` field can be one single pattern, like `exports="*"`, or several inclusion patterns. For example, if we have some python code that we want the recipe to use in a `helpers.py` file, and have some text file, `info.txt`, we want to read and display during the recipe evaluation we would do something like:

```python
exports = "helpers.py", "info.txt"
```

Exclude patterns are also possible, with the ! prefix:
This is an optional attribute, only to be used if the package recipe requires these other files for evaluation of the recipe.

**exports_sources**

There are 2 ways of getting source code to build a package. Using the `source()` recipe method and using the `exports_sources` field. With `exports_sources` you specify which sources are required, and they will be exported together with the `conanfile.py`, copying them from your folder to the local conan cache. Using `exports_sources` the package recipe can be self-contained, containing the source code like in a snapshot, and then not requiring downloading or retrieving the source code from other origins (git, download) with the `source()` method when it is necessary to build from sources.

The `exports_sources` field can be one single pattern, like `exports_sources="*"`, or several inclusion patterns. For example, if we have the source code inside “include” and “src” folders, and there are other folders that are not necessary for the package recipe, we could do:

```python
exports_sources = "include*", "src*
```

Exclude patterns are also possible, with the `!` prefix:

```python
exports_sources = "include*", "src*", "!src/build/*
```

This is an optional attribute, used typically when `source()` is not specified. The main difference with `exports` is that `exports` files are always retrieved (even if pre-compiled packages exist), while `exports_sources` files are only retrieved when it is necessary to build a package from sources.

**generators**

Generators specify which is the output of the `install` command in your project folder. By default, a `conanbuildinfo.txt` file is generated, but you can specify different generators and even use more than one.

```python
class MyLibConan(ConanFile):
    generators = "cmake", "gcc"
```

Check the full `generators list`.

**should_configure, should_build, should_install, should_test**

Read only variables defaulted to `True`.

These variables allow you to control the build stages of a recipe during a `conan build` command with the optional arguments `--configure/--build/--install/--test`. For example, consider this `build()` method:

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

If nothing is specified, all four methods will be called. But using command line arguments, this can be changed:
Autotools and Meson helpers already implement the same functionality. For other build systems, you can use these variables in the `build()` method:

```python
def build(self):
    if self.should_configure:
        # Run my configure stage
    if self.should_build:
        # Run my build stage
    if self.should_install:
        # If my build has install, otherwise use package()
        # Run my install stage
    if self.should_test:
        # Run my test stage
```

Note that the `should_configure`, `should_build`, `should_install`, `should_test` variables will always be `True` while building in the cache and can be only modified for the local flow with `conan build`.

### build_policy

With the `build_policy` attribute the package creator can change the default conan’s build behavior. The allowed `build_policy` values are:

- **missing**: If no binary package is found, Conan will build it without the need to invoke `conan install --build missing` option.
- **always**: The package will be built always, *retrieving each time the source code* executing the “source” method.

```python
class PocoTimerConan(ConanFile):
    build_policy = "always" # "missing"
```

### short_paths

This attribute is specific to Windows, and ignored on other operating systems. It tells Conan to workaround the limitation of 260 chars in Windows paths.

**Important**: Since Windows 10 (ver. 10.0.14393), it is possible to enable long paths at the system level. Latest python 2.x and 3.x installers enable this by default. With the path limit removed both on the OS and on Python, the `short_paths` functionality becomes unnecessary, and may be disabled explicitly through the `CONAN_USER_HOME_SHORT` environment variable.

Enabling short paths management will “link” the `source` and `build` directories of the package to the drive root, something like `C:\.conan\tmpdir`. All the folder layout in the local cache is maintained.

Set `short_paths=True` in your `conanfile.py`:
from conans import ConanFile
class ConanFileTest(ConanFile):
    ...
    short_paths = True

See also:
There is an environment variable CONAN_USE_ALWAYS_SHORT_PATHS to globally enable this behavior for all packages.

no_copy_source

The attribute no_copy_source tells the recipe that the source code will not be copied from the source folder to the build folder. This is mostly an optimization for packages with large source codebases, to avoid extra copies. It is mandatory that the source code must not be modified at all by the configure or build scripts, as the source code will be shared among all builds.

To be able to use it, the package recipe can access the self.source_folder attribute, which will point to the build folder when no_copy_source=False or not defined, and will point to the source folder when no_copy_source=True

When this attribute is set to True, the self.copy() lines will be called twice, one copying from the source folder and the other copying from the build folder.

source_folder

The folder in which the source code lives.

When a package is built in the Conan local cache its value is the same as the build folder by default. This is due to the fact that the source code is copied from the source folder to the build folder to ensure isolation and avoiding modifications of shared common source code among builds for different configurations. Only when no_copy_source=True this folder will actually point to the package source folder in the local cache.

When executing Conan commands in the Package development flow like conan source, this attribute will be pointing to the folder specified in the command line.

install_folder

The folder in which the installation of packages outputs the generator files with the information of dependencies. By default in the the local cache its value is the same as self.build_folder one.

When executing Conan commands in the Package development flow like conan install or conan build, this attribute will be pointing to the folder specified in the command line.

build_folder

The folder used to build the source code. In the local cache a build folder is created with the name of the package ID that will be built.

When executing Conan commands in the Package development flow like conan build, this attribute will be pointing to the folder specified in the command line.
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.

When executing Conan commands in the Package development flow like conan package, this attribute will be pointing to the folder specified in the command line.

cpp_info

Important: This attribute is only defined inside package_info() method being None elsewhere.

The self.cpp_info is responsible for storing all the information needed by consumers of a package: include directories, library names, library paths... There are some default values that will be applied automatically if not indicated otherwise.

This object should be filled in package_info() method.

<table>
<thead>
<tr>
<th>NAME</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>self.cpp_info.includedirs</td>
<td>Ordered list with include paths. Defaulted to &quot;include&quot;</td>
</tr>
<tr>
<td>self.cpp_info.libdirs</td>
<td>Ordered list with lib paths. Defaulted to &quot;lib&quot;</td>
</tr>
<tr>
<td>self.cpp_info.resdirs</td>
<td>Ordered list of resource (data) paths. Defaulted to &quot;res&quot;</td>
</tr>
<tr>
<td>self.cpp_info.bindirs</td>
<td>Ordered list with include paths. Defaulted to &quot;bin&quot;</td>
</tr>
<tr>
<td>self.cpp_info.builddirs</td>
<td>Ordered list with build scripts directory paths. Defaulted to &quot;&quot; (Package folder directory) CMake generators will search in these dirs for files like findXXX.cmake</td>
</tr>
<tr>
<td>self.cpp_info.libs</td>
<td>Ordered list with the library names, Defaulted to [] (empty)</td>
</tr>
<tr>
<td>self.cpp_info.defines</td>
<td>Preprocessor definitions. Defaulted to [] (empty)</td>
</tr>
<tr>
<td>self.cpp_info.cflags</td>
<td>Ordered list with pure C flags. Defaulted to [] (empty)</td>
</tr>
<tr>
<td>self.cpp_info.cxxflags</td>
<td>[DEPRECATED: use cxxflags instead]</td>
</tr>
<tr>
<td>self.cpp_info.sharedlinkflags</td>
<td>Ordered list with linker flags (shared libs). Defaulted to [] (empty)</td>
</tr>
<tr>
<td>self.cpp_info.exelinkflags</td>
<td>Ordered list with linker flags (executables). Defaulted to [] (empty)</td>
</tr>
<tr>
<td>self.cpp_info.rootpath</td>
<td>Filled with the root directory of the package, see deps_cpp_info</td>
</tr>
</tbody>
</table>

The paths of the directories in the directory variables indicated above are relative to the self.package_folder directory.

See also:

Read package_info() for more info.
**deps_cpp_info**

Contains the `cpp_info` object of the requirements of the recipe. In addition of the above fields, there are also properties to obtain the absolute paths:

<table>
<thead>
<tr>
<th>NAME</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>self.cpp_info.include_paths</td>
<td>Same as <code>includedirs</code> but transformed to absolute paths</td>
</tr>
<tr>
<td>self.cpp_info.lib_paths</td>
<td>Same as <code>libdirs</code> but transformed to absolute paths</td>
</tr>
<tr>
<td>self.cpp_info.bin_paths</td>
<td>Same as <code>bindirs</code> but transformed to absolute paths</td>
</tr>
<tr>
<td>self.cpp_info.build_paths</td>
<td>Same as <code>builddirs</code> but transformed to absolute paths</td>
</tr>
<tr>
<td>self.cpp_info.res_paths</td>
<td>Same as <code>resdirs</code> but transformed to absolute paths</td>
</tr>
</tbody>
</table>

To get a list of all the dependency names from `deps_cpp_info`, you can call the `deps` member:

```python
class PocoTimerConan(ConanFile):
    ...
    def build(self):
        # deps is a list of package names: ['Poco', 'zlib', 'OpenSSL']
        deps = self.deps_cpp_info.deps
```

It can be used to get information about the dependencies, like used compilation flags or the root folder of the package:

```python
class PocoTimerConan(ConanFile):
    ...
    requires = "zlib/1.2.11@conan/stable", "OpenSSL/1.0.2l@conan/stable"
    ...
    def build(self):
        # Get the directory where zlib package is installed
        self.deps_cpp_info['zlib'].rootpath

        # Get the absolute paths to zlib include directories (list)
        self.deps_cpp_info['zlib'].include_paths

        # Get the sharedlinkflags property from OpenSSL package
        self.deps_cpp_info['OpenSSL'].sharedlinkflags
```

**env_info**

This attribute is only defined inside `package_info()` method, being None elsewhere, so please use it only inside this method.

The `self.env_info` object can be filled with the environment variables to be declared in the packages reusing the recipe.

See also:

Read `package_info()` method docs for more info.

**deps_env_info**

You can access to the declared environment variables of the requirements of the recipe.

Note: The environment variables declared in the requirements of a recipe are automatically applied and it can be accessed with the python `os.environ` dictionary. Nevertheless if you want to access to the variable declared by some specific requirement you can use the `self.deps_env_info` object.
import os

class RecipeConan(ConanFile):
    ...
    requires = "package1/1.0@conan/stable", "package2/1.2@conan/stable"
    ...
    
def build(self):
        # Get the SOMEVAR environment variable declared in the "package1"
        self.deps_env_info["package1"].SOMEVAR

        # Access to the environment variables globally
        os.environ["SOMEVAR"]

user_info

This attribute is only defined inside package_info() method, being None elsewhere, so please use it only inside this method.

The self.user_info object can be filled with any custom variable to be accessed in the packages reusing the recipe.

See also:
Read package_info() method docs for more info.

deps_user_info

You can access the declared user_info.XXX variables of the requirements through the self.deps_user_info object like this:

import os

class RecipeConan(ConanFile):
    ...
    requires = "package1/1.0@conan/stable"
    ...
    
def build(self):
        self.deps_user_info["package1"].SOMEVAR

info

Object used to control the unique ID for a package. Check the package_id() to see the details of the self.info object.

apply_env

When True (Default), the values from self.deps_env_info (corresponding to the declared env_info in the requires and build_requires) will be automatically applied to the os.environ.

Disable it setting apply_env to False if you want to control by yourself the environment variables applied to your recipes.
You can apply manually the environment variables from the requires and build_requires:

```python
import os
from conans import tools

class RecipeConan(ConanFile):
    apply_env = False

    def build(self):
        with tools.environment_append(self.env):
            # The same if we specified apply_env = True
            pass
```

**in_local_cache**

A boolean attribute useful for conditional logic to apply in user folders local commands. It will return True if the conanfile resides in the local cache (we are installing the package) and False if we are running the conanfile in a user folder (local Conan commands).

```python
import os
class RecipeConan(ConanFile):
    ...

    def build(self):
        if self.in_local_cache:
            # we are installing the package
        else:
            # we are building the package in a local directory
```

**develop**

A boolean attribute useful for conditional logic. It will be True if the package is created with `conan create`, or if the `conanfile.py` is in user space:

```python
class RecipeConan(ConanFile):

    def build(self):
        if self.develop:
            self.output.info("Develop mode")
```

It can be used for conditional logic in other methods too, like `requirements()`, `package()`, etc. This recipe will output “Develop mode” if:

$ conan create . user/testing
# or
$ mkdir build && cd build && conan install ..
$ conan build ..

But it will not output that when it is a transitive requirement or installed with `conan install`. 
**keep_imports**

Just before the `build()` method is executed, if the conanfile has an `imports()` method, it is executed into the build folder, to copy binaries from dependencies that might be necessary for the `build()` method to work. After the method finishes, those copied (imported) files are removed, so they are not later unnecessarily repackaged.

This behavior can be avoided declaring the `keep_imports=True` attribute. This can be useful, for example to repackage artifacts.

**scm**

**Warning:** This is an experimental feature subject to breaking changes in future releases.

Used to clone/checkou a repository. It is a dictionary with the following possible values:

```python
from conans import ConanFile, CMake, tools

class HelloConan(ConanFile):
    scm = {
        "type": "git",
        "subfolder": "hello",
        "url": "https://github.com/conan-io/hello.git",
        "revision": "master"
    }
```

- **type** (Required): Currently only `git` and `svn` are supported. Others can be added eventually.
- **url** (Required): URL of the remote or `auto` to capture the remote from the local working copy (credentials will be removed from it). When type is `svn` it can contain the `peg_revision`.
- **revision** (Required): id of the revision or `auto` to capture the current working copy one. When type is `git`, it can also be the branch name or a tag.
- **subfolder** (Optional, Defaulted to `.`): A subfolder where the repository will be cloned.
- **username** (Optional, Defaulted to `None`): When present, it will be used as the login to authenticate with the remote.
- **password** (Optional, Defaulted to `None`): When present, it will be used as the password to authenticate with the remote.
- **verify_ssl** (Optional, Defaulted to `True`): Verify SSL certificate of the specified `url`.
- **submodule** (Optional, Defaulted to `None`):
  - shallow: Will sync the git submodules using `submodule sync`
  - recursive: Will sync the git submodules using `submodule sync --recursive`

SCM attributes are evaluated in the workspace context where the `conanfile.py` is located before exporting it to the Conan cache, so these values can be returned from arbitrary functions that depend on the workspace layout. Nevertheless, all the other code in the recipe must be able to run in the export folder inside the cache, where it has access only to the files exported (see attribute `exports`) and to any other functionality from a `python_requires`.

To know more about the usage of `scm` check:

- Creating packages/Recipe and sources in a different repo
• Creating packages/Recipe and sources in the same repo

revision_mode

**Warning:** This attribute is part of the package revisions feature, so it is also an experimental feature subject to breaking changes in future releases.

This attribute allow each recipe to declare how the revision for the recipe itself should be computed. It can take three 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.

python_requires

**Warning:** This attribute is part of the python requires feature, so it is also an experimental feature subject to breaking changes in future releases.

Python requires are associated with the ConanFile declared in the recipe file, data from those imported recipes is accessible using the python_requires attribute in the recipe itself. This attribute is a dictionary where the key is the name of the python requires reference and the value is a dictionary with the following information:

- **ref**: full reference of the python requires.
- **exports_folder**: directory in the cache where the exported files are located.
- **exports_sources_folder**: directory in the cache where the files exported using the exports_sources attribute of the python requires recipe are located.

You can use this information to copy files accompanying a python requires to the consumer workspace.:.

```python
from conans import ConanFile

class PyReq(ConanFile):
    name = "pyreq"
    exports_sources = "CMakeLists.txt"

    def source(self):
        pyreq = self.python_requires['pyreq']
        self.copy("CMakeLists.txt", src=pyreq.exports_sources_folder, dst=self.source_folder)
```

14.3.2 Methods

source()

Method used to retrieve the source code from any other external origin like github using $ git clone or just a regular download.
For example, “exporting” the source code files, together with the `conanfile.py` file, can be handy if the source code is not under version control. But if the source code is available in a repository, you can directly get it from there:

```python
from conans import ConanFile
class HelloConan(ConanFile):
    name = "Hello"
    version = "0.1"
    settings = "os", "compiler", "build_type", "arch"
    def source(self):
        self.run("git clone https://github.com/conan-io/hello.git")
        # You can also change branch, commit or whatever
        # self.run("cd hello && git checkout 2fe5...")
        # Or using the Git class:
        # git = tools.Git(folder="hello")
        # git.clone("https://github.com/conan-io/hello.git")
```

This will work, as long as git is in your current path (so in Win you probably want to run things in msysgit, cmder, etc). You can also use another VCS or direct download/unzip. For that purpose, we have provided some helpers, but you can use your own code or origin as well. This is a snippet of the conanfile of the Poco library:

```python
from conans import ConanFile
from conans.tools import download, unzip, check_md5, check_sha1, check_sha256
import os
import shutil

class PocoConan(ConanFile):
    name = "Poco"
    version = "1.6.0"
    def source(self):
        zip_name = "poco-1.6.0-release.zip"
        download("https://github.com/pocoproject/poco/archive/poco-1.6.0-release.zip", zip_name)
        # check_md5(zip_name, "51e11f2c02a36689d6ed655b6ff9ec9")
        # check_sha1(zip_name, "8d87812ce591ced8ce3a022beec1df1c8b2fac87")
        # check_sha256(zip_name, "653f983c30974d292de58444626884bee84a2731989ff5a336b93a0fef168d79")
        unzip(zip_name)
        shutil.move("poco-poco-1.6.0-release", "poco")
        os.unlink(zip_name)
```

The download, unzip utilities can be imported from conan, but you can also use your own code here to retrieve source code from any origin. You can even create packages for pre-compiled libraries you already have, even if you don’t have the source code. You can download the binaries, skip the `build()` method and define your `package()` and `package_info()` accordingly.

You can also use `check_md5()`, `check_sha1()` and `check_sha256()` from the `tools` module to verify that a package is downloaded correctly.

**Note:** It is very important to recall that the `source()` method will be executed just once, and the source code will be shared for all the package builds. So it is not a good idea to conditionally use settings or options to make changes or patches on the source code. Maybe the only setting that makes sense is the OS `self.settings.os`, if not doing cross-building, for example to retrieve different sources:
If you need to patch the source code or build scripts differently for different variants of your packages, you can do it in the `build()` method, which uses a different folder and source code copy for each variant.

### Build helpers

You can use these classes to prepare your build system’s command invocation:

- **CMake**: Prepares the invocation of `cmake` command with your settings.

- **AutoToolsBuildEnvironment**: If you are using configure/Makefile to build your project you can use this helper. Read more: Building with Autotools.

- **MSBuild**: If you are using Visual Studio compiler directly to build your project you can use this helper `MSBuild()`. For lower level control, the `VisualStudioBuildEnvironment` can also be used: `VisualStudioBuildEnvironment`.

### (Unit) Testing your library

We have seen how to run package tests with conan, but what if we want to run full unit tests on our library before packaging, so that they are run for every build configuration? Nothing special is required here. We can just launch the tests from the last command in our `build()` method:

```python
def build(self):
    cmake = CMake(self)
    cmake.configure()
    cmake.build()
    # here you can run CTest, launch your binaries, etc
    cmake.test()
```

### package()

The actual creation of the package, once that it is built, is done in the `package()` method. Using the `self.copy()` method, artifacts are copied from the build folder to the package folder.

The syntax of `self.copy` inside `package()` is as follows:
self.copy(pattern, dst="", src="", keep_path=True, symlink=None, excludes=None, _,
→ignore_case=False)

Parameters:

- **pattern** (Required): A pattern following fnmatch syntax of the files you want to copy, from the build to
  the package folders. Typically something like *.lib or *.h.
- **src** (Optional, Defaulted to ""): The folder where you want to search the files in the build folder. If
  you know that your libraries when you build your package will be in build/lib, you will typically use
  build/lib in this parameter. Leaving it empty means the root build folder in local cache.
- **dst** (Optional, Defaulted to ""): Destination folder in the package. They will typically be include for
  headers, lib for libraries and so on, though you can use any convention you like. Leaving it empty means
  the root package folder in local cache.
- **keep_path** (Optional, Defaulted to True): Means if you want to keep the relative path when you copy the
  files from the src folder to the dst one. Typically headers are packaged with relative path.
- **symlinks** (Optional, Defaulted to None): Set it to True to activate symlink copying, like typical lib.so->lib.so.9.
- **excludes** (Optional, Defaulted to None): Single pattern or a tuple of patterns to be excluded from the
  copy. If a file matches both the include and the exclude pattern, it will be excluded.
- **ignore_case** (Optional, Defaulted to False): If enabled, it will do a case-insensitive pattern matching.

For example:

```python
self.copy("*.h", "include", "build/include") #keep_path default is True
```

The final path in the package will be: include/mylib/path/header.h, and as the include is usually added to
the path, the includes will be in the form: #include "mylib/path/header.h" which is something desired.

```python
keep_path=False is something typically desired for libraries, both static and dynamic. Some compilers as MSVC,
put them in paths as Debug/x64/MyLib/Mylib.lib. Using this option, we could write:
```

```python
self.copy("*.lib", "lib", ",", keep_path=False)
```

And it will copy the lib to the package folder lib/Mylib.lib, which can be linked easily.

**Note:** If you are using CMake and you have an install target defined in your CMakeLists.txt, you might be able to
reuse it for this package() method. Please check How to reuse cmake install for package() method.

This method copies files from build/source folder to the package folder depending on two situations:

- **Build folder and source folder are the same**: Normally during conan create source folder content is
copied to the build folder. In this situation src parameter of self.copy() will be relative to the build folder
in the local cache.

- **Build folder is different from source folder**: When developing a package recipe and source and build folder
are different (conan package --source-folder=source --build-folder=build) or when
no_copy_source is defined, every self.copy() is internally called twice: One will copy from the source
folder (src parameter of self.copy() will point to the source folder), and the other will copy from the
build folder (src parameter of self.copy() will point to the build folder).
package_info()
cpp_info

Each package has to specify certain build information for its consumers. This can be done in the cpp_info attribute within the package_info() method.

The cpp_info attribute has the following properties you can assign/append to:

```python
self.cpp_info.includedirs = ['include']  # Ordered list of include paths
self.cpp_info.libs = []                 # The libs to link against
self.cpp_info.libdirs = ['lib']         # Directories where libraries can be found
self.cpp_info.resdirs = ['res']         # Directories where resources, data, etc can be found
self.cpp_info.bindirs = ['bin']         # Directories where executables and shared libs can be found
self.cpp_info.srcdirs = []              # Directories where sources can be found (debugging, reusing sources)
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
```

- **includedirs**: List of relative paths (starting from the package root) of directories where headers can be found. By default it is initialized to ['include'], and it is rarely changed.
- **libs**: Ordered list of libs the client should link against. Empty by default, it is common that different configurations produce different library names. For example:

```python
def package_info(self):
    if not self.settings.os == "Windows":
        self.cpp_info.libs = ["libzmq-static.a"] if self.options.static else ["libzmq.so"]
    else:
        ...
```

- **libdirs**: List of relative paths (starting from the package root) of directories in which to find library object binaries (*.lib, *.a, *.so, *.dylib). By default it is initialized to ['lib'], and it is rarely changed.
- **resdirs**: List of relative paths (starting from the package root) of directories in which to find resource files (images, xml, etc). By default it is initialized to ['res'], and it is rarely changed.
- **bindirs**: List of relative paths (starting from the package root) of directories in which to find library runtime binaries (like Windows .dlls). By default it is initialized to ['bin'], and it is rarely changed.
- **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).
- **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:

```python
if self.options.static:
    if self.settings.compiler == "Visual Studio":
        self.cpp_info.libs.append("ws2_32")
```

(continues on next page)
self.cpp_info.defines = ["ZMQ_STATIC"]

if not self.settings.os == "Windows":
    self.cpp_info.cxxflags = ["-pthread"]

Note that due to the way that some build systems, like CMake, manage forward and back slashes, it might be more robust passing flags for Visual Studio compiler with dash instead. Using "/NODEFAULTLIB:MSVCR70", for example, might fail when using CMake targets mode, so the following is preferred and works both in the global and targets mode of CMake:

def package_info(self):
    self.cpp_info.exelinkflags = ["-NODEFAULTLIB:MSVCR70",
                               "-DEFAULTLIB:LIBCMT"]

If your recipe has requirements, you can access to your requirements cpp_info as well using the deps_cpp_info object.

class OtherConan(ConanFile):
    name = "OtherLib"
    version = "1.0"
    requires = ["MyLib/1.6.0@conan/stable"

    def build(self):
        self.output.warn(self.deps_cpp_info["MyLib"].libdirs)

Note: Please take into account that defining self.cpp_info.bindirs directories, does not have any effect on system paths, PATH environment variable, nor will be directly accessible by consumers. self.cpp_info information is translated to build-systems information via generators, for example for CMake, it will be a variable in conanbuildinfo.cmake. If you want a package to make accessible its executables to its consumers, you have to specify it with self.env_info as described in env_info.

env_info

Each package can also define some environment variables that the package needs to be reused. It’s specially useful for installer packages, to set the path with the “bin” folder of the packaged application. This can be done in the env_info attribute within the package_info() method.

self.env_info.path.append("ANOTHER VALUE")  # Append "ANOTHER VALUE" to the path variable
self.env_info.othervar = "OTHER VALUE"  # Assign "OTHER VALUE" to the othervar variable
self.env_info.thirdvar.append("some value")  # Every variable can be set or appended a new value

One of the most typical usages for the PATH environment variable, would be to add the current binary package directories to the path, so consumers can use those executables easily:

# assuming the binaries are in the "bin" subfolder
self.env_info.PATH.append(os.path.join(self.package_folder, "bin"))

The virtualenv generator will use the self.env_info variables to prepare a script to activate/deactivate a virtual environment. However, this could be directly done using the virtualrunenv generator.
They will be automatically applied before calling the consumer *conanfile.py* methods `source()`, `build()`, `package()` and `imports()`.

If your recipe has requirements, you can access to your requirements `env_info` as well using the `deps_env_info` object.

```python
class OtherConan(ConanFile):
    name = "OtherLib"
    version = "1.0"
    requires = "MyLib/1.6.0@conan/stable"

    def build(self):
        self.output.warn(self.deps_env_info["MyLib"].othervar)
```

**user_info**

If you need to declare custom variables not related with C/C++ (`cpp_info`) and the variables are not environment variables (`env_info`), you can use the `self.user_info` object.

Currently only the `cmake`, `cmake_multi` and `txt` generators support `user_info` variables.

```python
class MyLibConan(ConanFile):
    name = "MyLib"
    version = "1.6.0"

    # ...

    def package_info(self):
        self.user_info.var1 = 2
```

For the example above, in the `cmake` and `cmake_multi` generators, a variable `CONAN_USER_MYLIB_var1` will be declared. If your recipe has requirements, you can access to your requirements `user_info` using the `deps_user_info` object.

```python
class OtherConan(ConanFile):
    name = "OtherLib"
    version = "1.0"
    requires = "MyLib/1.6.0@conan/stable"

    def build(self):
        self.out.warn(self.deps_user_info["MyLib"].var1)
```

**Important**: Both `env_info` and `user_info` objects store information in a “key <-> value” form and the values are always considered strings. This is done for serialization purposes to `conanbuildinfo.txt` files and to avoid the deserialization of complex structures. It is up to the consumer to convert the string to the expected type:

```python
# In a dependency
self.user_info.jars="jar1.jar, jar2.jar, jar3.jar"  # Use a string, not a list
...

# In the dependent conanfile
jars = self.deps_user_info["Pkg"].jars
jar_list = jars.replace(" ", """).split(",")
```
configure(), config_options()

If the package options and settings are related, and you want to configure either, you can do so in the `configure()` and `config_options()` methods.

```python
class MyLibConan(ConanFile):
    name = "MyLib"
    version = "2.5"
    settings = "os", "compiler", "build_type", "arch"
    options = {"static": [True, False],
               "header_only": [True, False]}

    def configure(self):
        # If header only, the compiler, etc, does not affect the package!
        if self.options.header_only:
            self.settings.clear()
            self.options.remove("static")
```

The package has 2 options set, to be compiled as a static (as opposed to shared) library, and also not to involve any builds, because header-only libraries will be used. In this case, the settings that would affect a normal build, and even the other option (static vs shared) do not make sense, so we just clear them. That means, if someone consumes MyLib with the `header_only=True` option, the package downloaded and used will be the same, irrespective of the OS, compiler or architecture the consumer is building with.

You can also restrict the settings used deleting any specific one. For example, it is quite common for C libraries to delete the `compiler.libcxx` and `compiler.cppstd` as your library does not depend on any C++ standard library:

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

The most typical usage would be the one with `configure()` while `config_options()` should be used more sparingly. `config_options()` is used to configure or constraint the available options in a package, before they are given a value. So when a value is tried to be assigned it will raise an error. For example, let’s suppose that a certain package library cannot be built as shared library in Windows, it can be done:

```python
def config_options(self):
    if self.settings.os == "Windows":
        del self.options.shared
```

This will be executed before the actual assignment of options (then, such options values cannot be used inside this function), so the command `conan install -o Pkg:shared=True` will raise an exception in Windows saying that `shared` is not an option for such package.

Invalid configuration

Conan allows the recipe creator to declare invalid configurations, those that are known not to work with the library being packaged. There is an especial kind of exception that can be raised from the `configure()` method to state this situation: `conans.errors.ConanInvalidConfiguration`. Here it is an example of a recipe for a library that doesn’t support Windows operating system:

```python
def configure(self):
    if self.settings.os != "Windows":
        raise ConanInvalidConfiguration("Library MyLib is only supported for Windows")
```
This exception will be propagated and Conan application will finish with a *special return code*.

**requirements()**

Besides the `requires` field, more advanced requirement logic can be defined in the `requirements()` optional method, using for example values from the package `settings` or `options`:

```python
def requirements(self):
    if self.options.myoption:
        self.requires("zlib/1.2@drl/testing")
    else:
        self.requires("opencv/2.2@drl/stable")
```

This is a powerful mechanism for handling **conditional dependencies**.

When you are inside the method, each call to `self.requires()` will add the corresponding requirement to the current list of requirements. It also has optional parameters that allow defining the special cases, as is shown below:

```python
def requirements(self):
    self.requires("zlib/1.2@drl/testing", private=True, override=False)
```

**self.requires() parameters:**

- **override** (Optional, Defaulted to `False`): True means that this is not an actual requirement, but something to be passed upstream and override possible existing values.

- **private** (Optional, Defaulted to `False`): True means that this requirement will be somewhat embedded (like a static lib linked into a shared lib), so it is not required to link.

**Note:** To prevent accidental override of transitive dependencies, check the config variable `general.error_on_override` or the environment variable `CONAN_ERROR_ON_OVERRIDE`.

**build_requirements()**

Build requirements are requirements that are only installed and used when the package is built from sources. If there is an existing pre-compiled binary, then the build requirements for this package will not be retrieved.

This method is useful for defining conditional build requirements, for example:

```python
class MyPkg(ConanFile):
    def build_requirements(self):
        if self.settings.os == "Windows":
            self.build_requires("ToolWin/0.1@user/stable")
```

See also:

*Build requirements*

**system_requirements()**

It is possible to install system-wide packages from conan. Just add a `system_requirements()` method to your `conanfile` and specify what you need there.
For a special use case you can use also `conans.tools.os_info` object to detect the operating system, version and distribution (linux):

- `os_info.is_linux`: True if Linux.
- `os_info.is_windows`: True if Windows.
- `os_info.is_macos`: True if macOS.
- `os_info.is_freebsd`: True if FreeBSD.
- `os_info.is_solaris`: True if SunOS.
- `os_info.os_version`: OS version.
- `os_info.os_version_name`: Common name of the OS (Windows 7, Mountain Lion, Wheezy...).
- `os_info.linux_distro`: Linux distribution name (None if not Linux).
- `os_info.bash_path`: Returns the absolute path to a bash in the system.
- `os_info.uname(options=None)`: Runs the “uname” command and returns the output. You can pass arguments with the `options` parameter.
- `os_info.detect_windows_subsystem()`: Returns “MSYS”, “MSYS2”, “CYGWIN” or “WSL” if any of these Windows subsystems are detected.

You can also use `SystemPackageTool` class, that will automatically invoke the right system package tool: `apt`, `yum`, `pkg`, `pkgutil`, `brew` and `pacman` depending on the system we are running.

```python
from conans.tools import os_info, SystemPackageTool

def system_requirements(self):
    pack_name = None
    if os_info.linux_distro == "ubuntu":
        if os_info.os_version > "12":
            pack_name = "package_name_in_ubuntu_10"
        else:
            pack_name = "package_name_in_ubuntu_12"
    elif os_info.linux_distro == "fedora" or os_info.linux_distro == "centos":
        pack_name = "package_name_in_fedora_and_centos"
    elif os_info.is_macos:
        pack_name = "package_name_in_macos"
    elif os_info.is_freebsd:
        pack_name = "package_name_in_freebsd"
    elif os_info.is_solaris:
        pack_name = "package_name_in_solaris"
    
    if pack_name:
        installer = SystemPackageTool()
        installer.install(pack_name)  # Install the package, will update the package database if pack_name isn't already installed
```

On Windows, there is no standard package manager, however `choco` can be invoked as an optional:

```python
from conans.tools import os_info, SystemPackageTool, ChocolateyTool

def system_requirements(self):
    if os_info.is_windows:
        pack_name = "package_name_in_windows"
```

(continues on next page)
SystemPackageTool

```
def SystemPackageTool(tool=None)
```

Available tool classes: **AptTool, YumTool, BrewTool, PkgTool, PkgUtilTool, ChocolateyTool, PacManTool**.

**Methods:**

- **update()**: Updates the system package manager database. It’s called automatically from the `install()` method by default.

- **install(packages, update=True, force=False)**: Installs the `packages` (could be a list or a string). If `update` is True it will execute `update()` first if it’s needed. The packages won’t be installed if they are already installed at least of `force` parameter is set to True. If `packages` is a list the first available package will be picked (short-circuit like logical `or`). **Note:** This list of packages is intended for providing **alternative** names for the same package, to account for small variations of the name for the same package in different distros. To install different packages, one call to `install()` per package is necessary.

The use of `sudo` in the internals of the `install()` and `update()` methods is controlled by the `CONAN_SYSREQUIRES_SUDO` environment variable, so if the users don’t need sudo permissions, it is easy to opt-in/out.

When the environment variable `CONAN_SYSREQUIRES_SUDO` is not defined, Conan will try to use `sudo` if the following conditions are met:

- *sudo* is available in the `PATH`.
- The platform name is `posix` and the UID (user id) is not 0

Conan will keep track of the execution of this method, so that it is not invoked again and again at every Conan command. The execution is done per package, since some packages of the same library might have different system dependencies. If you are sure that all your binary packages have the same system requirements, just add the following line to your method:

```
def system_requirements(self):
    self.global_system_requirements=True
    if ...
```

**imports()**

Importing files copies files from the local store to your project. This feature is handy for copying shared libraries (`dylib` in Mac, `dll` in Win) to the directory of your executable, so that you don’t have to mess with your `PATH` to run them. But there are other use cases:

- Copy an executable to your project, so that it can be easily run. A good example is the Google’s `protobuf` code generator.

- Copy package data to your project, like configuration, images, sounds... A good example is the OpenCV demo, in which face detection XML pattern files are required.

Importing files is also very convenient in order to redistribute your application, as many times you will just have to bundle your project’s bin folder.
A typical imports() method for shared libs could be:

```python
def imports(self):
    self.copy("*.dll", "", "bin")
    self.copy("*.dylib", "", "lib")
```

The self.copy() method inside imports() supports the following arguments:

```python
def copy(pattern, dst="", src="", root_package=None, folder=False, ignore_case=False,
       excludes=None, keep_path=True)
```

Parameters:

- `pattern` (Required): An fnmatch file pattern of the files that should be copied.
- `dst` (Optional, Defaulted to ""): Destination local folder, with reference to current directory, to which the files will be copied.
- `src` (Optional, Defaulted to ""): Source folder in which those files will be searched. This folder will be stripped from the dst parameter. E.g., `lib/Debug/x86`
- `root_package` (Optional, Defaulted to all packages in deps): An fnmatch pattern of the package name ("OpenCV", "Boost") from which files will be copied.
- `folder` (Optional, Defaulted to False): If enabled, it will copy the files from the local cache to a subfolder named as the package containing the files. Useful to avoid conflicting imports of files with the same name (e.g. License).
- `ignore_case` (Optional, Defaulted to False): If enabled, it will do a case-insensitive pattern matching.
- `excludes` (Optional, Defaulted to None): Allows defining a list of patterns (even a single pattern) to be excluded from the copy, even if they match the main pattern.
- `keep_path` (Optional, Defaulted to True): Means if you want to keep the relative path when you copy the files from the src folder to the dst one. Useful to ignore (keep_path=False) path of library.dll files in the package it is imported from.

Example to collect license files from dependencies:

```python
def imports(self):
    self.copy("license *", dst="licenses", folder=True, ignore_case=True)
```

If you want to be able to customize the output user directory to work with both the cmake and cmake_multi generators, then you can do:

```python
def imports(self):
    dest = os.getenv("CONAN_IMPORT_PATH", "bin")
    self.copy("*.dll", dst=dest, src="bin")
    self.copy("*.dylib*", dst=dest, src="lib")
```

And then use, for example: `conan install . -e CONAN_IMPORT_PATH=Release -g cmake_multi`

When a conanfile recipe has an imports() method and it builds from sources, it will do the following:

- Before running build() it will execute imports() in the build folder, copying dependencies artifacts
- Run the build() method, which could use such imported binaries.
- Remove the copied (imported) artifacts after build() is finished.

You can use the keep_imports attribute to keep the imported artifacts, and maybe repackage them.
**package_id()**

Creates a unique ID for the package. Default package ID is calculated using settings, options and requires properties. When a package creator specifies the values for any of those properties, it is telling that any value change will require a different binary package.

However, sometimes a package creator would need to alter the default behavior, for example, to have only one binary package for several different compiler versions. In that case you can set a custom `self.info` object implementing this method and the package ID will be computed with the given information:

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

Please, check the section *Defining Package ABI Compatibility* to get more details.

**self.info**

This `self.info` object stores the information that will be used to compute the package ID.

This object can be manipulated to reflect the information you want in the computation of the package ID. For example, you can delete any setting or option:

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

**self.info.header_only()**

The package will always be the same, irrespective of the OS, compiler or architecture the consumer is building with.

```python
def package_id(self):
    self.info.header_only()
```

**self.info.vs_toolset_compatible() / self.info.vs_toolset_incompatible()**

By default (vs_toolset_compatible() mode) Conan will generate the same binary package when the compiler is Visual Studio and the compiler.toolset matches the specified compiler.version. For example, if we install some packages specifying the following settings:

```python
def package_id(self):
    self.info.vs_toolset_compatible()
    # self.info.vs_toolset_incompatible()
```

```python
compiler="Visual Studio"
compiler.version=14
```

And then we install again specifying these settings:

```python
compiler="Visual Studio"
compiler.version=15
compiler.toolset=v140
```
The compiler version is different, but Conan will not install a different package, because the used toolchain in both cases are considered the same. You can deactivate this default behavior using calling `self.info.vc_toolset_incompatible()`.

This is the relation of Visual Studio versions and the compatible toolchain:

<table>
<thead>
<tr>
<th>Visual Studio Version</th>
<th>Compatible toolset</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>v141</td>
</tr>
<tr>
<td>14</td>
<td>v140</td>
</tr>
<tr>
<td>12</td>
<td>v120</td>
</tr>
<tr>
<td>11</td>
<td>v110</td>
</tr>
<tr>
<td>10</td>
<td>v100</td>
</tr>
<tr>
<td>9</td>
<td>v90</td>
</tr>
<tr>
<td>8</td>
<td>v80</td>
</tr>
</tbody>
</table>

`self.info.discard_build_settings()` / `self.info.include_build_settings()`

By default (`discard_build_settings()`) Conan will generate the same binary when you change the `os_build` or `arch_build` when the `os` and `arch` are declared respectively. This is because `os_build` represent the machine running Conan, so, for the consumer, the only setting that matters is where the built software will run, not where is running the compilation. The same applies to `arch_build`.

With `self.info.include_build_settings()`, Conan will generate different packages when you change the `os_build` or `arch_build`.

```python
def package_id(self):
    self.info.discard_build_settings()
    # self.info.include_build_settings()
```

`self.info.default_std_matching()` / `self.info.default_std_non_matching()`

By default (`default_std_matching()`) Conan will detect the default C++ standard of your compiler to not generate different binary packages.

For example, you already built some `gcc 6.1` packages, where the default std is `gnu14`. If you specify a value for the setting `compiler.cppstd` equal to the default one, `gnu14`, Conan won’t generate new packages, because it was already the default of your compiler.

With `self.info.default_std_non_matching()`, Conan will generate different packages when you specify the compiler.cppstd even if it matches with the default of the compiler being used:

```python
def package_id(self):
    self.info.default_std_non_matching()
    # self.info.default_std_matching()
```

Same behavior applies if you use the deprecated setting `cppstd`.

`build_id()`

In the general case, there is one build folder for each binary package, with the exact same hash/ID of the package. However this behavior can be changed, there are a couple of scenarios that this might be interesting:
• You have a build script that generates several different configurations at once, like both debug and release artifacts, but you actually want to package and consume them separately. Same for different architectures or any other setting.

• You build just one configuration (like release), but you want to create different binary packages for different consuming cases. For example, if you have created tests for the library in the build step, you might want to create two packages: one just containing the library for general usage, and another one also containing the tests. First package could be used as a reference and the other one as a tool to debug errors.

In both cases, if using different settings, the system will build twice (or more times) the same binaries, just to produce a different final binary package. With the `build_id()` method this logic can be changed. `build_id()` will create a new package ID/hash for the build folder, and you can define the logic you want in it. For example:

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

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

So this recipe will generate a final different package for each debug/release configuration. But as the `build_id()` will generate the same ID for any `build_type`, then just one folder and one build will be done. Such build should build both debug and release artifacts, and then the `package()` method should package them accordingly to the `self.settings.build_type` value. Different builds will still be executed if using different compilers or architectures. This method is basically an optimization of build time, avoiding multiple re-builds.

Other information like custom package options can also be changed:

```python
def build_id(self):
    self.info_build.options.myoption = 'MyValue' # any value possible
    self.info_build.options.fullsource = 'Always'
```

If the `build_id()` method does not modify the `build_id`, and produce a different one than the `package_id`, then the standard behavior will be applied. Consider the following:

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

def build_id(self):
    if self.settings.os == "Windows":
        self.info_build.settings.build_type = "Any"
```

This will only produce a build ID different if the package is for Windows. So the behavior in any other OS will be the standard one, as if the `build_id()` method was not defined: the build folder will be wiped at each `conan create` command and a clean build will be done.

**deploy()**

This method can be used in a `conanfile.py` to install in the system or user folder artifacts from packages.

```python
def deploy(self):
    self.copy("*.exe")  # copy from current package
    self.copy_deps("*.dll") # copy from dependencies
```

Where:

- `self.copy()` is the `self.copy()` method executed inside `package()` method.
- `self.copy_deps()` is the same as `self.copy()` method inside `imports()` method.
Both methods allow the definition of absolute paths (to install in the system), in the \texttt{dst} argument. By default, the \texttt{dst} destination folder will be the current one.

The \texttt{deploy()} method is designed to work on a package that is installed directly from its reference, as:

\begin{verbatim}
$ conan install Pkg/0.1@user/channel
> ... Pkg/0.1@user/testing deploy(): Copied 1 '.dll' files: mylib.dll
> Pkg/0.1@user/testing deploy(): Copied 1 '.exe' files: myexe.exe
\end{verbatim}

All other packages and dependencies, even transitive dependencies of \texttt{“Pkg/0.1@user/testing”} will not be deployed, it is the responsibility of the installed package to deploy what it needs from its dependencies.

### 14.3.3 Python requires

It is possible to reuse python code existing in other \texttt{conanfile.py} recipes with the \texttt{python_requires()} functionality, doing something like:

\begin{verbatim}
from conans import python_requires
base = python_requires("MyBuild/0.1@user/channel")

class PkgTest(base.MyBase):
    ...
    def build(self):
        base.my_build(self.settings)
\end{verbatim}

See this section: \textit{Python requires: reusing python code in recipes}

### 14.3.4 Output and Running

**Output contents**

Use the \texttt{self.output} to print contents to the output.

\begin{verbatim}
self.output.success("This is a good, should be green")
self.output.info("This is a neutral, should be white")
self.output.warn("This is a warning, should be yellow")
self.output.error("Error, should be red")
self.output.rewrite_line("for progress bars, issues a cr")
\end{verbatim}

Check the source code. You might be able to produce different outputs with different colors.

**Running commands**

\begin{verbatim}
run(self, command, output=True, cwd=None, win_bash=False, subsystem=None, msys_
    →mingw=True, ignore_errors=False, run_environment=False, with_login=True):
\end{verbatim}

\texttt{self.run()} is a helper to run system commands and throw exceptions when errors occur, so that command errors are not pass unnoticed. It is just a wrapper for \texttt{os.system()}.

When the environment variable \texttt{CONAN_PRINT_RUN_COMMANDS} is set to true (or its equivalent \texttt{print_run_commands conan.conf} configuration variable, under [general]) then all the invocations of \texttt{self.run()} will print to output the command to be executed.

14.3. \texttt{conanfile.py}
Optional parameters:

- **output** (Optional, Defaulted to `True`) When True it will write in stdout. You can pass any stream that accepts a write method like a `six.StringIO()`:
  
  ```python
  from six import StringIO  # Python 2 and 3 compatible
  mybuf = StringIO()
  self.run("mycommand", output=mybuf)
  self.output.warn(mybuf.getvalue())
  ```

- **cwd** (Optional, Defaulted to . current directory): Current directory to run the command.

- **win_bash** (Optional, Defaulted to `False`): When True, it will run the configure/make commands inside a bash.

- **subsystem** (Optional, Defaulted to `None` will autodetect the subsystem): Used to escape the command according to the specified subsystem.

- **msys_mingw** (Optional, Defaulted to `True`) If the specified subsystem is MSYS2, will start it in MinGW mode (native windows development).

- **ignore_errors** (Optional, Defaulted to `False`). This method raises an exception if the command fails. If `ignore_errors=True`, it will not raise an exception. Instead, the user can use the return code to check for errors.

- **run_environment** (Optional, Defaulted to `False`). Applies a RunEnvironment, so the environment variables PATH, LD_LIBRARY_PATH, and DYLIB_LIBRARY_PATH are defined in the command execution adding the values of the “lib” and “bin” folders of the dependencies. Allows executables to be easily run using shared libraries from its dependencies.

- **with_login** (Optional, Defaulted to `True`): Pass the --login flag to bash command when using `win_bash` parameter. This might come handy when you don’t want to create a fresh user session for running the command.

## 14.4 Generators

Generators are specific components that provide the information of dependencies calculated by Conan in a suitable format for a build system. They normally provide Conan users with a `conanbuildinfo.XXX` file that can be included or injected to the specific build system. The file generated contains information of dependencies in form of different variables and sometimes function helpers too.

You can specify a generator in:

- The `[generators]` section from `conanfile.txt`.

- The `generators` attribute in `conanfile.py`.

- The command line when installing dependencies `conan install --generator`.

Available generators:

### 14.4.1 cmake

This is the reference page for cmake generator. Go to Integrations/CMake if you want to learn how to integrate your project or recipes with CMake.

It generates a file named `conanbuildinfo.cmake` and declares some variables and methods.
Variables in `conanbuildinfo.cmake`

**• Package declared variables:**

For each requirement `conanbuildinfo.cmake` file declares the following variables. `XXX` is the name of the require in uppercase. e.g. “ZLIB” for `zlib/1.2.8@lasote/stable` requirement:

<table>
<thead>
<tr>
<th>NAME</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONAN_XXX_ROOT</td>
<td>Abs path to root package folder.</td>
</tr>
<tr>
<td>CONAN_INCLUDE_DIRS_XXX</td>
<td>Header’s folders</td>
</tr>
<tr>
<td>CONAN_LIB_DIRS_XXX</td>
<td>Library folders (default <code>{CONAN_XXX_ROOT}/lib</code>)</td>
</tr>
<tr>
<td>CONAN_BIN_DIRS_XXX</td>
<td>Binary folders (default <code>{CONAN_XXX_ROOT}/bin</code>)</td>
</tr>
<tr>
<td>CONAN_SRC_DIRS_XXX</td>
<td>Sources folders</td>
</tr>
<tr>
<td>CONAN_LIBS_XXX</td>
<td>Library names to link</td>
</tr>
<tr>
<td>CONAN_DEFINES_XXX</td>
<td>Library defines</td>
</tr>
<tr>
<td>CONAN_COMPILE_DEFINITIONS_XXX</td>
<td>Compile definitions</td>
</tr>
<tr>
<td>CONAN_CXX_FLAGS_XXX</td>
<td>CXX flags</td>
</tr>
<tr>
<td>CONAN_SHARED_LINK_FLAGS_XXX</td>
<td>Shared link flags</td>
</tr>
<tr>
<td>CONAN_C_FLAGS_XXX</td>
<td>C flags</td>
</tr>
</tbody>
</table>

**• Global declared variables:**

This generator also declares some global variables with the aggregated values of all our requirements. The values are ordered in the right order according to the dependency tree.

<table>
<thead>
<tr>
<th>NAME</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONAN_INCLUDE_DIRS</td>
<td>Aggregated header’s folders</td>
</tr>
<tr>
<td>CONAN_LIB_DIRS</td>
<td>Aggregated library folders</td>
</tr>
<tr>
<td>CONAN_BIN_DIRS</td>
<td>Aggregated binary folders</td>
</tr>
<tr>
<td>CONAN_SRC_DIRS</td>
<td>Aggregated sources folders</td>
</tr>
<tr>
<td>CONAN_LIBS</td>
<td>Aggregated library names to link</td>
</tr>
<tr>
<td>CONAN_DEFINES</td>
<td>Aggregated library defines</td>
</tr>
<tr>
<td>CONAN_COMPILE_DEFINITIONS</td>
<td>Aggregated compile definitions</td>
</tr>
<tr>
<td>CONAN_CXX_FLAGS</td>
<td>Aggregated CXX flags</td>
</tr>
<tr>
<td>CONAN_SHARED_LINK_FLAGS</td>
<td>Aggregated Shared link flags</td>
</tr>
<tr>
<td>CONAN_C_FLAGS</td>
<td>Aggregated C flags</td>
</tr>
</tbody>
</table>

**• User information declared variables:**

If any of the requirements is filling the `user_info` object in the `package_info` method a set of variables will be declared following this naming:

<table>
<thead>
<tr>
<th>NAME</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONAN_USER_XXXX_YYYY</td>
<td>User declared value</td>
</tr>
</tbody>
</table>

Where `XXXX` means the name of the requirement in uppercase and `YYYY` the variable name. For example, if this recipe declares:

```python
class MyLibConan(ConanFile):
    name = "MyLib"
    version = "1.6.0"
    # ...
```

(continues on next page)
def package_info(self):
    self.user_info.var1 = 2

Other library requiring MyLib and using this generator will get:

Listing 8: conanbuildinfo.cmake

```cmake
# ...
set(CONAN_USER_MYLIB_var1 "2")
```

Macros available in `conanbuildinfo.cmake`

**conan_basic_setup()**

This is a helper and general purpose macro that uses all the macros below to set all the CMake variables according to the Conan generated variables. See the macros below for detailed information.

```cmake
macro(conan_basic_setup)
    set(options TARGETS NO_OUTPUT_DIRS SKIP_RPATH KEEP_RPATHS SKIP_STD SKIP_FPIC)
```

Parameters:

- **TARGETS** (Optional): Setup all the CMake variables by target (only CMake > 3.1.2). Activates the call to the macro `conan_target_link_libraries()`.
- **NO_OUTPUT_DIRS** (Optional): Do not adjust the output directories. Deactivates the call to the macro `conan_output_dirs_setup()`.
- **SKIP_RPATH** (Optional): [DEPRECATED] Use `KEEP_RPATHS` instead. Activate `CMAKE_SKIP_RPATH` variable in OSX.
- **KEEP_RPATHS** (Optional): Do not adjust the `CMAKE_SKIP_RPATH` variable in OSX. Activates the call to the macro `conan_set_rpath()`.
- **SKIP_STD** (Optional): Do not adjust the C++ standard flag in `CMAKE_CXX_FLAGS`. Deactivates the call to the macro `conan_set_std()`.
- **SKIP_FPIC** (Optional): Do not adjust the `CMAKE_POSITION_INDEPENDENT_CODE` flag. Deactivates the call to the macro `conan_set_fpic()`.

**Note:** You can also call each of the following macros individually instead of using the `conan_basic_setup()`.

**conan_target_link_libraries()**

Helper to link all libraries to a specified target.

These targets are:

- A **CONAN_PKG::PkgName** target per package in the dependency graph. This is an **IMPORTED** **INTERFACE** target. **IMPORTED** because it is external, a pre-compiled library. **INTERFACE**, because it doesn’t necessarily match a library, it could be a header-only library, or the package could even contain several libraries. It contains all the properties (include paths, compile flags, etc.) that are defined in the `package_info()` method of the recipe.
• Inside each package a CONAN_LIB::PkgName_LibName target will be generated for each library. Its type is IMPORTED UNKNOWN and its main purpose is to provide a correct link order. Their only properties are the location and the dependencies.

• A CONAN_PKG depends on every CONAN_LIB that belongs to it, and to its direct public dependencies (e.g. other CONAN_PKG targets from its requirements).

• Each CONAN_LIB depends on the direct public dependencies CONAN_PKG targets of its container package. This guarantees correct link order.

conan_check_compiler()

Checks that your compiler matches the one declared in settings.

conan_output_dirs_setup()

Adjust the bin/ and lib/ output directories.

conan_set_find_library_paths()

Set CMAKE_INCLUDE_PATH and CMAKE_INCLUDE_PATH.

conan_global_flags()

Set the corresponding variables to CMake’s include_directories() and link_directories().

conan_define_targets()

Define the targets for each dependency (target flags instead of global flags).

conan_set_rpath()

Set CMAKE_SKIP_RPATH=1 in the case of working in OSX.

conan_set_vs_runtime()

Adjust the runtime flags /MD, /MDd, /MT or /MTd for Visual Studio.

conan_set_std()

Set CMAKE_CXX_STANDARD and CMAKE_CXX_EXTENSIONS to the appropriate values.

conan_set_libcxx()

Adjust the standard library flags (libc++, libstdc++, libstdc++11) in CMAKE_CXX_FLAGS.
conan_set_find_paths()

Adjust CMAKE_MODULE_PATH and CMAKE_PREFIX_PATH to the values of deps_cpp_info.build_paths.

Input variables for conanbuildinfo.cmake

CONAN_CMAKE_SILENT_OUTPUT

Default to: FALSE

Activate it to silence the Conan message output.

14.4.2 cmake_multi

This is the reference page for cmake_multi generator. Go to Integrations/CMake if you want to learn how to integrate your project or recipes with CMake.

This generator will create 3 files with the general information and specific Debug/Release ones:

- `conanbuildinfo_release.cmake`: Variables adjusted only for build type Release
- `conanbuildinfo_debug.cmake`: Variables adjusted only for build type Debug
- `conanbuildinfo_multi.cmake`: Which includes the other two and enables its use and has more generic variables and macros.

Variables in conanbuildinfo_release.cmake

Same as `conanbuildinfo.cmake` with suffix _RELEASE

Variables in conanbuildinfo_debug.cmake

Same as `conanbuildinfo.cmake` with suffix _DEBUG

Macros available in conanbuildinfo_multi.cmake

conan_basic_setup()

This is a helper and general purpose macro that uses all the macros below to set all the CMake variables according to the Conan generated variables. See the macros below for detailed information.

```python
macro(conan_basic_setup)
  set(options TARGETS NO_OUTPUT_DIRS SKIP_RPATH KEEP_RPATHS SKIP_STD SKIP_FPIC)
endmacro(conan_basic_setup)
```

Parameters:

- **TARGETS** (Optional): Setup all the CMake variables by target (only CMake > 3.1.2). Activates the call to the macro `conan_target_link_libraries()`.
- **NO_OUTPUT_DIRS** (Optional): This variable has no effect and it works as if it was activated by default (does not set fixed output directories and uses the default ones designated by CMake).
- **SKIP_RPATH** (Optional): [DEPRECATED] Use KEEP_RPATHS instead. Activate CMAKE_SKIP_RPATH variable in OSX.
• KEEP_RPATHS (Optional): Do not adjust the CMAKE_SKIP_RPATH variable in OSX. Activates the call to the macro conan_set_rpath().

• SKIP_STD (Optional): Do not adjust the C++ standard flag in CMAKE_CXX_FLAGS. Deactivates the call to the macro conan_set_std().

• SKIP_FPICT (Optional): Do not adjust the CMAKE_POSITION_INDEPENDENT_CODE flag. Deactivates the call to the macro conan_set_fpic().

Note: You can also call each of the following macros individually instead of using the conan_basic_setup().

conan_target_link_libraries()

Helper to link all libraries to a specified target.

These targets are:

• A CONAN_PKG::PkgName target per package in the dependency graph. This is an IMPORTED INTERFACE target. IMPORTED because it is external, a pre-compiled library. INTERFACE, because it doesn’t necessarily match a library, it could be a header-only library, or the package could even contain several libraries. It contains all the properties (include paths, compile flags, etc.) that are defined in the package_info() method of the recipe.

• Inside each package a CONAN_LIB::PkgName_LibName target will be generated for each library. Its type is IMPORTED UNKNOWN and its main purpose is to provide a correct link order. Their only properties are the location and the dependencies.

• A CONAN_PKG depends on every CONAN_LIB that belongs to it, and to its direct public dependencies (e.g. other CONAN_PKG targets from its requirements).

• Each CONAN_LIB depends on the direct public dependencies CONAN_PKG targets of its container package. This guarantees correct link order.

conan_check_compiler()

Checks that your compiler matches the one declared in settings.

conan_output_dirs_setup()

Adjust the bin/ and lib/ output directories.

conan_global_flags()

Set the corresponding variables to CMake’s include_directories() and link_directories().

conan_define_targets()

Define the targets for each dependency (target flags instead of global flags).
conan Documentation, Release 1.16.0

conan_set_rpath()

Set CMAKE_SKIP_RPATH=1 in the case of working in OSX.

conan_set_vs_runtime()

Adjust the runtime flags /MD, /MDd, /MT or /MTd for Visual Studio.

conan_set_std()

Set CMAKE_CXX_STANDARD and CMAKE_CXX_EXTENSIONS to the appropriate values.

conan_set_libcxx()

Adjust the standard library flags (libc++, libstdc++, libstdc++11) in CMAKE_CXX_FLAGS.

conan_set_find_paths()

Adjust CMAKE_MODULE_PATH and CMAKE_PREFIX_PATH to the values of deps_cpp_info.build_paths.

Input variables for conanbuildinfo_multi.cmake

CONAN_CMAKE_SILENT_OUTPUT

Default to: FALSE

Activate it to silence the Conan message output.

14.4.3 cmake_paths

This is the reference page for cmake_paths generator. Go to Integrations/CMake if you want to learn how to integrate your project or recipes with CMake.

It generates a file named conan_paths.cmake and declares two variables:

Variables in conan_paths.cmake

<table>
<thead>
<tr>
<th>NAME</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMAKE_MODULE_PATH</td>
<td>Containing all requires root folders, any declared self.cpp_info.builddirs and the current directory of this file</td>
</tr>
<tr>
<td>CMAKE_PREFIX_PATH</td>
<td>Containing all requires root folders, any declared self.cpp_info.builddirs and the current directory of this file</td>
</tr>
<tr>
<td>CONAN_XXX_ROOT</td>
<td>For each dep, the root folder, being XXX the dep name uppercase. Useful when a .cmake is patched with cmake.patch_config_paths()</td>
</tr>
</tbody>
</table>
14.4.4 cmake_find_package

This is the reference page for cmake_find_package generator. Go to Integrations/CMake if you want to learn how to integrate your project or recipes with CMake.

The cmake_find_package generator creates a file for each requirement specified in the conanfile.

The name of the files follow the pattern Find<package_name>.cmake. So for the zlib/1.2.11@conan/stable package, a Findzlib.cmake file will be generated.

Variables in Find{name}.cmake

Being {name} the package name:

<table>
<thead>
<tr>
<th>NAME</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>{name}_FOUND</td>
<td>Set to 1</td>
</tr>
<tr>
<td>{name}_VERSION</td>
<td>Package version</td>
</tr>
<tr>
<td>{name}_INCLUDE_DIRS</td>
<td>Containing all the include directories of the package</td>
</tr>
<tr>
<td>{name}_INCLUDES</td>
<td>Same as the XXX_INCLUDE_DIRS</td>
</tr>
<tr>
<td>{name}_DEFINITIONS</td>
<td>Definitions of the library</td>
</tr>
<tr>
<td>{name}_LIBRARIES</td>
<td>Library paths to link</td>
</tr>
<tr>
<td>{name}_LIBS</td>
<td>Same as XXX_LIBRARIES</td>
</tr>
</tbody>
</table>

Target in Find<package_name>.cmake

A target named {name}::{name} target is generated with the following properties adjusted:

- INTERFACE_INCLUDE_DIRECTORIES: Containing all the include directories of the package.
- INTERFACE_LINK_LIBRARIES: Library paths to link.
- INTERFACE_COMPILE_DEFINITIONS: Definitions of the library.

The targets are transitive. So, if your project depends on packages A and B, and at the same time A depends on C, the A target will contain automatically the properties of the C dependency, so in your CMakeLists.txt file you only need to find_package(A) and find_package(B).

14.4.5 cmake_find_package_multi

Warning: This is an experimental feature subject to breaking changes in future releases.

This is the reference page for cmake_find_package_multi generator. Go to Integrations/CMake if you want to learn how to integrate your project or recipes with CMake.

Generated files

For each conan package in your graph, it will generate 2 files and 1 more per different build_type. Being {name} the package name:
### Targets

A target named `{name}::{name}` target is generated with the following properties adjusted:

- **INTERFACE_INCLUDE_DIRECTORIES**: Containing all the include directories of the package.
- **INTERFACE_LINK_LIBRARIES**: Library paths to link.
- **INTERFACE_COMPILE_DEFINITIONS**: Definitions of the library.

The targets contains multi-configuration properties, for example, the compile options property is declared like this:

```cmake
set_property(TARGET {name}::{name} PROPERTY INTERFACE_COMPILE_OPTIONS
  $<$<CONFIG:Release>:${{{name}_COMPILE_OPTIONS_RELEASE_LIST}}>
  $<$<CONFIG:RelWithDebInfo>:${{{name}_COMPILE_OPTIONS_RELWITHDEBINFO_LIST}}>
  $<$<CONFIG:MinSizeRel>:${{{name}_COMPILE_OPTIONS_MINSIZEREL_LIST}}>
  $<$<CONFIG:Debug>:${{{name}_COMPILE_OPTIONS_DEBUG_LIST}}>)
```

The targets are also transitive. So, if your project depends on packages A and B, and at the same time A depends on C, the A target will contain automatically the properties of the C dependency, so in your `CMakeLists.txt` file you only need to `find_package(A)` and `find_package(B)`.

You also need to adjust `CMAKE_PREFIX_PATH` and `CMAKE_MODULE_PATH` so CMake can locate all the `{name}Config.cmake` files: The `CMAKE_PREFIX_PATH` is used by the `find_package` and the `CMAKE_MODULE_PATH` is used by the `find_dependency` calls that locates the transitive dependencies.

#### 14.4.6 visual_studio

This is the reference page for `visual_studio` generator. Go to `Integrations/Visual Studio` if you want to learn how to integrate your project or recipes with Visual Studio.

Generates a file named `conanbuildinfo.props` containing an XML that can be imported to your Visual Studio project.

Generated xml structure:

```xml
<?xml version="1.0" encoding="utf-8"?>
<Project ToolsVersion="4.0" xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <ImportGroup Label="PropertySheets" />
  <PropertyGroup Label="UserMacros" />
  <PropertyGroup Label="Conan-RootDirs">
    <Conan-Lib1-Root>{PACKAGE LIB1 FOLDER}</Conan-Poco-Root>
  </PropertyGroup>
</Project>
```

(continues on next page)
Note that for single-configuration packages, which is the most typical, conan install Debug/Release, 32/64bits, packages separately. So a different property sheet will be generated for each configuration. The process could be:

Given for example a conanfile.txt like:

```
[requires]
Pkg/0.1@user/channel

[generators]
visual_studio
```

And assuming that binary packages exist for Pkg/0.1@user/channel, we could do:

```
$ mkdir debug32 && cd debug32
$ conan install .. -s compiler="Visual Studio" -s compiler.version=15 -s arch=x86 -s build_type=Debug
$ cd ..
$ mkdir debug64 && cd debug64
$ conan install .. -s compiler="Visual Studio" -s compiler.version=15 -s arch=x86_64 -s build_type=Debug
$ cd ..
$ mkdir release32 && cd release32
$ conan install .. -s compiler="Visual Studio" -s compiler.version=15 -s arch=x86 -s build_type=Release
$ cd ..
$ mkdir release64 && cd release64
```

(continues on next page)
### 14.4.7 visual_studio_multi

This is the reference page for `visual_studio_multi` generator. Go to [Integrations/Visual Studio](#) if you want to learn how to integrate your project or recipes with Visual Studio.

#### Usage

```bash
$ conan install . -g visual_studio_multi -s arch=x86 -s build_type=Debug
$ conan install . -g visual_studio_multi -s arch=x86 -s build_type=Debug
$ conan install . -g visual_studio_multi -s arch=x86 -s build_type=Release
$ conan install . -g visual_studio_multi -s arch=x86 -s build_type=Release
```

These commands will generate 5 files for each compiler version:

- `conanbuildinfo_multi.props`: All properties
- `conanbuildinfo_release_x64_v141.props.props`: Variables for release/64bits/VS2015 (toolset v141)
- `conanbuildinfo_debug_x64_v141.props.props`: Variables for debug/64bits/VS2015 (toolset v141)
- `conanbuildinfo_release_win32_v141.props.props`: Variables for release/32bits/VS2015 (toolset v141)
- `conanbuildinfo_debug_win32_v141.props.props`: Variables for debug/32bits/VS2015 (toolset v141)

You can now load `conanbuildinfo_multi.props` in your Visual Studio IDE property manager, and all configurations will be loaded at once.

Each one of the configurations will have the format and information defined in [the `visual_studio` generator](#).

### 14.4.8 visual_studio_legacy

Generates a file named `conanbuildinfo.vsprops` containing an XML that can be imported to your Visual Studio 2008 project. Note that the format of this file is different and incompatible with the `conanbuildinfo.props` file generated with the `visual_studio` generator for newer VS.

Generated xml structure:
This file can be loaded from the Menu->View->PropertyManager window, selecting “Add Existing Property Sheet” for the desired configuration.

Note that for single-configuration packages, which is the most typical, conan install Debug and Release packages separately. So a different property sheet will be generated for each configuration. The process could be:
Given for example a conanfile.txt like:

```
[requires]
Pkg/0.1@user/channel

[generators]
visual_studio_legacy
```

And assuming that binary packages exist for Pkg/0.1@user/channel, we could do:

```
$ mkdir debug && cd debug
$ conan install .. -s compiler="Visual Studio" -s compiler.version=9 -s arch=x86 -s build_type=Debug
$ cd ..
$ mkdir release && cd release
$ conan install .. -s compiler="Visual Studio" -s compiler.version=9 -s arch=x86 -s build_type=Release
# Now go to VS 2008 Property Manager, load the respective sheet into each configuration
```

The above process can be simplified using profiles (assuming you have created “vs9release” profile), and you can also specify the generators in the command line:

```
$ conan install .. -pr=vs9release -g visual_studio_legacy
```

## 14.4.9 xcode

This is the reference page for xcode generator. Go to Integrations/Xcode if you want to learn how to integrate your project or recipes with Xcode.

The xcode generator creates a file named conanbuildinfo.xcconfig that can be imported to your Xcode project.

The file declare these variables:

<table>
<thead>
<tr>
<th>VARIABLE</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>HEADER_SEARCH_PATHS</td>
<td>The requirements include dirs</td>
</tr>
<tr>
<td>LIBRARY_SEARCH_PATHS</td>
<td>The requirements lib dirs</td>
</tr>
<tr>
<td>OTHER_LDFLAGS</td>
<td>-lXXX corresponding to library names</td>
</tr>
<tr>
<td>GCC_PREPROCESSOR_DEFINITIONS</td>
<td>The requirements definitions</td>
</tr>
<tr>
<td>OTHER_CFLAGS</td>
<td>The requirements cflags</td>
</tr>
<tr>
<td>OTHER_CPLUSPLUSFLAGS</td>
<td>The requirements cxxflags</td>
</tr>
<tr>
<td>FRAMEWORK_SEARCH_PATHS</td>
<td>The requirements root folders, so xcode can find packaged frameworks</td>
</tr>
</tbody>
</table>

## 14.4.10 compiler_args

This is the reference page for compiler_args generator. Go to Integrations/Compilers on command line if you want to learn how to integrate your project calling your compiler in the command line.

Generates a file named conanbuildinfo.args containing a command line parameters to invoke gcc, clang or cl compiler.

You can use the compiler_args generator directly to build simple programs:

gcc/clang:
> g++ timer.cpp @conanbuildinfo.args -o bin/timer

cl:

$ cl /EHsc timer.cpp @conanbuildinfo.args

gcc/clang

<table>
<thead>
<tr>
<th>FLAG</th>
<th>MEANING</th>
</tr>
</thead>
<tbody>
<tr>
<td>-DXXX</td>
<td>Corresponding to requirements defines</td>
</tr>
<tr>
<td>-IXXX</td>
<td>Corresponding to requirements include dirs</td>
</tr>
<tr>
<td>-Wl,-rpathXXX</td>
<td>Corresponding to requirements lib dirs</td>
</tr>
<tr>
<td>-LXXX</td>
<td>Corresponding to requirements libs</td>
</tr>
<tr>
<td>-m64</td>
<td>For x86_64 architecture</td>
</tr>
<tr>
<td>-m32</td>
<td>For x86 architecture</td>
</tr>
<tr>
<td>-DNDEBUG</td>
<td>For Release builds</td>
</tr>
<tr>
<td>-s</td>
<td>For Release builds (only gcc)</td>
</tr>
<tr>
<td>-g</td>
<td>For Debug builds</td>
</tr>
<tr>
<td>-D_GLIBCXX_USE_CXX11_ABI=0</td>
<td>When setting libcxx == “libstdc++”</td>
</tr>
<tr>
<td>-D_GLIBCXX_USE_CXX11_ABI=1</td>
<td>When setting libcxx == “libstdc++11”</td>
</tr>
<tr>
<td>Other flags</td>
<td>cxxflags, cflags, sharedlinkflags, exelinkflags (applied directly)</td>
</tr>
</tbody>
</table>

cl (Visual Studio)

<table>
<thead>
<tr>
<th>FLAG</th>
<th>MEANING</th>
</tr>
</thead>
<tbody>
<tr>
<td>/DXXX</td>
<td>Corresponding to requirements defines</td>
</tr>
<tr>
<td>/IXXX</td>
<td>Corresponding to requirements include dirs</td>
</tr>
<tr>
<td>/LIBPATH:XX</td>
<td>Corresponding to requirements lib dirs</td>
</tr>
<tr>
<td>/MT, /MTd, /MD, /MDd</td>
<td>Corresponding to Runtime</td>
</tr>
<tr>
<td>-DNDEBUG</td>
<td>For Release builds</td>
</tr>
<tr>
<td>/Zi</td>
<td>For Debug builds</td>
</tr>
</tbody>
</table>

You can also use it in a recipe:

```
from conans import ConanFile

class PocoTimerConan(ConanFile):
    settings = "os", "compiler", "build_type", "arch"
    requires = "Poco/1.9.0@pocoproject/stable"
    generators = "compiler_args"
    default_options = {"Poco:shared": True, "OpenSSL:shared": True}

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

    def build(self):
        self.run("mkdir -p bin")
        command = 'g++ timer.cpp @conanbuildinfo.args -o bin/timer'
        self.run(command)
```

14.4. Generators
14.4.11 gcc

Deprecated, use compiler_args generator instead.

14.4.12 Boost Build

**Caution:** This generator is deprecated in favor of the b2 generator. See generator b2.

The `boost-build` generator creates a file named `project-root.jam` that can be used with your Boost Build build system script.

The generated `project-root.jam` will generate several sections, and an alias `conan-deps` with the sections name:

```plaintext
lib ssl :
  : # requirements
  <name>ssl
  <search>/path/to/package/227fb0ea22f4797212e72ba94ea89c7b3fbc2a0c/lib
  : # default-build
  : # usage-requirements
  <include>/path/to/package/227fb0ea22f4797212e72ba94ea89c7b3fbc2a0c/include

lib crypto :
  : # requirements
  <name>crypto
  <search>/path/to/package/227fb0ea22f4797212e72ba94ea89c7b3fbc2a0c/lib
  : # default-build
  : # usage-requirements
  <include>/path/to/package/227fb0ea22f4797212e72ba94ea89c7b3fbc2a0c/include

lib z :
  : # requirements
  <name>z
  <search>/path/to/package/8018a4df6e7d2b4630a814fa40c81b85b9182d2b/lib
  : # default-build
  : # usage-requirements
  <include>/path/to/package/8018a4df6e7d2b4630a814fa40c81b85b9182d2b/include

alias conan-deps :
  ssl
crypto
  z
```

14.4.13 B2 [EXPERIMENTAL]

This is the reference page for the B2, aka Boost Build, generator. It is a multi-generator to match the multi-build nature of B2.
Warning: This is an experimental feature subject to breaking changes in future releases.

Usage

Usage requirements for you to make use of the headers in that package. For example, for the above conanfile.txt the targets would be..

clara subproject:
alias libs
  : # source, none as it's header only
  : # requirements specific to the build
  ...
  : # default-build
  : # usage-requirements
  <include>/absolute/path/to/conan/package/include
  <define>...
  <cflags>...
  <cxxflags>...
  <link>shared:<linkflags>...
;

Where ... contains references to the variant specific constants. The target for boost_predef is equivalent as that's also a header only library. For libz it contains a built linkable library and hence it has additional targets for that.

libz subproject:

alias z
  : # source, no source as it's a searched pre-built library
  : # requirements
    <name>z
    <search>/absolute/path/to/conan/package/lib
    # rest of the requirements specific to the build
  : # default-build
  : # usage-requirements
    <include>/absolute/path/to/conan/package/include
    <define>...
    <cflags>...
    <cxxflags>...
    <link>shared:<linkflags>...
;

alias libs
  : # source
    z
  : # requirements specific to the build
  ...
  : # default-build
  : # usage-requirements
    <include>/absolute/path/to/conan/package/include
    <define>...
    <cflags>...
    <cxxflags>...
    <link>shared:<linkflags>...
;

**Constants in conanbuildinfo-XXX.jam**

This generator also defines constants, and path constants, in the project where the conanfile.txt is located. The constants define variant specific variables for all the packages and a transitive conan set of constants for all the packages.

- **Per package constants**

For each requirement conanbuildinfo-XXX.cmake file declares the following constants. variation is the name of the package and variation. That YYYY variation takes the form of a comma separated list of: package name,
address-model, architecture, target-os, toolset with version, and variant (debug, release, relwithdebinfo, and minsizerel). All are lower case and use the values of the corresponding B2 features. For example a boost_predef package dependency when building with apple-clang 9.0 and debug would be: boost_predef, 64,x86,darwin,clang-9.0,debug.

<table>
<thead>
<tr>
<th>NAME</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>rootpath(variation)</td>
<td>Abs path to root package folder.</td>
</tr>
<tr>
<td>includedirs(variation)</td>
<td>Header's folders</td>
</tr>
<tr>
<td>libdirs(variation)</td>
<td>Library folders (default {rootpath}/lib)</td>
</tr>
<tr>
<td>defines(variation)</td>
<td>Library defines</td>
</tr>
<tr>
<td>cppflags(variation)</td>
<td>CXX flags</td>
</tr>
<tr>
<td>sharedlinkflags(variation)</td>
<td>Shared link flags</td>
</tr>
<tr>
<td>cflags(variation)</td>
<td>C flags</td>
</tr>
<tr>
<td>requirements(variation)</td>
<td>B2 requirements</td>
</tr>
<tr>
<td>usage-requirements(variation)</td>
<td>B2 usage requirements</td>
</tr>
</tbody>
</table>

Both the requirements and usage-requirements are synthesized from the other constants.

- **Global declared constants**

  The generator also defines a corresponding set of constants that aggregate the values of all the package requirements. The constants for this are the same as the package-specific ones but with “conan” as the name of the project.

- **Constants from user_info**

  If any of the requirements is filling the user_info object in the package_info method a set of constants will be declared following this naming:

<table>
<thead>
<tr>
<th>NAME</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>user(name,variation)</td>
<td>User declared value</td>
</tr>
</tbody>
</table>

  variation is the package and variant as above and name the variable name in lower case. For example:

  ```
  class MyLibConan(ConanFile):
      name = "MyLib"
      version = "1.6.0"
      # ...
      def package_info(self):
          self.user_info.var1 = 2
  ```

  When other library requires MyLib and uses the B2 generator:

  conanbuildinfo-XXX.jam:

  ```
  constant user(var1,mylib,...) : "2" ;
  ```

14.4.14 qbs

This is the reference page for qbs generator. Go to Integrations/Qbs if you want to learn how to integrate your project or recipes with Qbs.

Generates a file named conanbuildinfo.qbs that can be used for your qbs builds.
A `Product ConanBasicSetup` contains the aggregated requirement values and also there is N `Product` declared, one per requirement.

```qbs
import qbs 1.0

Project {
    Product {
        name: "ConanBasicSetup"
        Export {
            Depends { name: "cpp" }
            cpp.includePaths: [{INCLUDE DIRECTORIES REQUIRE 1}, {INCLUDE DIRECTORIES REQUIRE 2}]
            cpp.libraryPaths: [{LIB DIRECTORIES REQUIRE 1}, {LIB DIRECTORIES REQUIRE 2}]
            cpp.systemIncludePaths: [{BIN DIRECTORIES REQUIRE 1}, {BIN DIRECTORIES REQUIRE 2}]
            cpp.dynamicLibraries: [{LIB NAMES REQUIRE 1}, {LIB NAMES REQUIRE 2}]
            cpp.defines: []
            cpp.cppFlags: []
            cpp.cFlags: []
            cpp.linkerFlags: []
        }
    }

    Product {
        name: "REQUIRE1"
        Export {
            Depends { name: "cpp" }
            cpp.includePaths: [{INCLUDE DIRECTORIES REQUIRE 1}]
            cpp.libraryPaths: [{LIB DIRECTORIES REQUIRE 1}]
            cpp.systemIncludePaths: [{BIN DIRECTORIES REQUIRE 1}]
            cpp.dynamicLibraries: [{LIB NAMES REQUIRE 1}]
            cpp.defines: []
            cpp.cppFlags: []
            cpp.cFlags: []
            cpp.linkerFlags: []
        }
        // lib root path: {ROOT PATH REQUIRE 1}
    }

    Product {
        name: "REQUIRE2"
        Export {
            Depends { name: "cpp" }
            cpp.includePaths: [{INCLUDE DIRECTORIES REQUIRE 2}]
            cpp.libraryPaths: [{LIB DIRECTORIES REQUIRE 2}]
            cpp.systemIncludePaths: [{BIN DIRECTORIES REQUIRE 2}]
            cpp.dynamicLibraries: [{LIB NAMES REQUIRE 2}]
            cpp.defines: []
            cpp.cppFlags: []
            cpp.cFlags: []
            cpp.linkerFlags: []
        }
        // lib root path: {ROOT PATH REQUIRE 2}
    }
}
```
14.4.15 qmake

This is the reference page for qmake generator. Go to Integrations/Qmake if you want to learn how to integrate your project or recipes with qmake.

Generates a file named conanbuildinfo.pri that can be used for your qbs builds. The file contains:

- N groups of variables, one group per require, declaring the same individual values: include_paths, libs, bin dirs, libraries, defines etc.
- One group of global variables with the aggregated values for all requirements.

Package declared vars

For each requirement conanbuildinfo.pri file declares the following variables. XXX is the name of the require in uppercase. e.k “ZLIB” for zlib/1.2.8@lasote/stable requirement:

<table>
<thead>
<tr>
<th>NAME</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONAN_XXX_ROOT</td>
<td>Abs path to root package folder.</td>
</tr>
<tr>
<td>CONAN_INCLUDEPATH_XXX</td>
<td>Header's folders</td>
</tr>
<tr>
<td>CONAN_LIB_DIRS_XXX</td>
<td>Library folders (default {CONAN_XXX_ROOT}/lib)</td>
</tr>
<tr>
<td>CONAN_BINDIRS_XXX</td>
<td>Binary folders (default {CONAN_XXX_ROOT}/bin)</td>
</tr>
<tr>
<td>CONAN_LIBS_XXX</td>
<td>Library names to link</td>
</tr>
<tr>
<td>CONAN_DEFINES_XXX</td>
<td>Library defines</td>
</tr>
<tr>
<td>CONAN_COMPILE_DEFINITIONS_XXX</td>
<td>Compile definitions</td>
</tr>
<tr>
<td>CONAN_QMAKE_CXXFLAGS_XXX</td>
<td>CXX flags</td>
</tr>
<tr>
<td>CONAN_QMAKE_LFLAGS_XXX</td>
<td>Shared link flags</td>
</tr>
<tr>
<td>CONAN_QMAKE_CFLAGS_XXX</td>
<td>C flags</td>
</tr>
</tbody>
</table>

Global declared vars

Conan also declares some global variables with the aggregated values of all our requirements. The values are ordered in the right order according to the dependency tree.

<table>
<thead>
<tr>
<th>NAME</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONAN_INCLUDEPATH</td>
<td>Aggregated header's folders</td>
</tr>
<tr>
<td>CONAN_LIB_DIRS</td>
<td>Aggregated library folders</td>
</tr>
<tr>
<td>CONAN_BINDIRS</td>
<td>Aggregated binary folders</td>
</tr>
<tr>
<td>CONAN_LIBS</td>
<td>Aggregated library names to link</td>
</tr>
<tr>
<td>CONAN_DEFINES</td>
<td>Aggregated library defines</td>
</tr>
<tr>
<td>CONAN_COMPILE_DEFINITIONS</td>
<td>Aggregated compile definitions</td>
</tr>
<tr>
<td>CONAN_QMAKE_CXXFLAGS</td>
<td>Aggregated CXX flags</td>
</tr>
<tr>
<td>CONAN_QMAKE_LFLAGS</td>
<td>Aggregated Shared link flags</td>
</tr>
<tr>
<td>CONAN_QMAKE_CFLAGS</td>
<td>Aggregated C flags</td>
</tr>
</tbody>
</table>

Methods available in conanbuildinfo.pri

<table>
<thead>
<tr>
<th>NAME</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>conan_basic_setup()</td>
<td>Setup all the qmake vars according to our settings with the global approach</td>
</tr>
</tbody>
</table>

14.4.16 scons

Conan provides integration with SCons with this generator.
The generated SConscript_conan will generate several dictionaries, like:

```
"conan" : {
    "CPPPATH" : ['/path/to/include'],
    "LIBPATH" : ['/path/to/lib'],
    "BINPATH" : ['/path/to/bin'],
    "LIBS" : ['hello'],
    "CPPDEFINES" : [],
    "CXXFLAGS" : [],
    "CCFLAGS" : [],
    "SHLINKFLAGS" : [],
    "LINKFLAGS" : [],
},

"Hello" : {
    "CPPPATH" : ['/path/to/include'],
    "LIBPATH" : ['/path/to/lib'],
    "BINPATH" : ['/path/to/bin'],
    "LIBS" : ['hello'],
    "CPPDEFINES" : [],
    "CXXFLAGS" : [],
    "CCFLAGS" : [],
    "SHLINKFLAGS" : [],
    "LINKFLAGS" : [],
}
```

The `conan` dictionary will contain the aggregated values for all dependencies, while the individual "Hello" dictionaries, one per package, will contain just the values for that specific dependency.

These dictionaries can be directly loaded into the environment like:

```
conan = SConscript('{}/SConscript_conan'.format(build_path_relative_to_sconstruct))
env.MergeFlags(conan['conan'])
```

### 14.4.17 pkg_config

Generates N files named `{dep_name}.pc`, containing a valid pkg-config file syntax. The `prefix` variable is automatically adjusted to the `package_folder`.

Go to Integrations/pkg-config and pc files/Use the pkg_config generator if you want to learn how to use this generator.

### 14.4.18 virtualenv

This is the reference page for virtualenv generator. Go to Mastering/Virtual Environments if you want to learn how to use conan virtual environments.

#### Created files

- `activate.{sh|bat|ps1}`
- `deactivate.{sh|bat|ps1}`
### Usage

Linux/macOS:

```bash
> source activate.sh
```

Windows:

```bash
> activate.bat
```

### Variables declared

<table>
<thead>
<tr>
<th>ENVIRONMENT VAR</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>PS1</td>
<td>New shell prompt value corresponding to the current directory name</td>
</tr>
<tr>
<td>OLD_PS1</td>
<td>Old PS1 value, to recover it in deactivation</td>
</tr>
<tr>
<td>XXXX</td>
<td>Any variable declared in the <code>self.env_info</code> object of the requirements.</td>
</tr>
</tbody>
</table>

### 14.4.19 virtualbuildenv

This is the reference page for `virtualbuildenv` generator. Go to [Mastering/Virtual Environments](#) if you want to learn how to use Conan virtual environments.

### Created files

- `activate_build.{sh|bat}
- `deactivate_build.{sh|bat}

### Usage

Linux/macOS:

```bash
$ source activate_build.sh
```

Windows:

```bash
$ activate_build.bat
```

### Variables declared

<table>
<thead>
<tr>
<th>ENVIRONMENT VAR</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIBS</td>
<td>Library names to link</td>
</tr>
<tr>
<td>LDFLAGS</td>
<td>Link flags, (-L, -m64, -m32)</td>
</tr>
<tr>
<td>CFLAGS</td>
<td>Options for the C compiler (-g, -s, -m64, -m32, -fPIC)</td>
</tr>
<tr>
<td>CXXFLAGS</td>
<td>Options for the C++ compiler (-g, -s, -stdlib, -m64, -m32, -fPIC)</td>
</tr>
<tr>
<td>CPPFLAGS</td>
<td>Preprocessor definitions (-D, -I)</td>
</tr>
<tr>
<td>LIB</td>
<td>Library paths separated with “;” (Visual Studio)</td>
</tr>
<tr>
<td>CL</td>
<td>“/F” flags with include directories (Visual Studio)</td>
</tr>
</tbody>
</table>
In the case of using this generator to compile with Visual Studio, it also sets the environment variables needed via 
`tools.vcvars()` to build your project. Some of these variables are:

```
VSINSTALLDIR=C:/Program Files (x86)/Microsoft Visual Studio/2017/Community/
WINDIR=C:/WINDOWS
WindowsLibPath=C:/Program Files (x86)/Windows Kits/10/UnionMetadata/10.0.16299.0;
WindowsSdkBinPath=C:/Program Files (x86)/Windows Kits/10/bin/
WindowsSdkDir=C:/Program Files (x86)/Windows Kits/10/
WindowsSDKLibVersion=10.0.16299.0/
WindowsSdkVerBinPath=C:/Program Files (x86)/Windows Kits/10/bin/10.0.16299.0/
```

### 14.4.20 virtualrunenv

This is the reference page for `virtualrunenv` generator. Go to *Mastering/Virtual Environments* if you want to
learn how to use conan virtual environments.

**Created files**

- `activate_run.{sh|bat}`
- `deactivate_run.{sh|bat}`

**Usage**

Linux/macOS:

```bash
> source activate_run.sh
```

Windows:

```bash
> activate_run.bat
```

**Variables declared**

<table>
<thead>
<tr>
<th>ENVIRONMENT VAR</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>PATH</td>
<td>With every <code>bin</code> folder of your requirements.</td>
</tr>
<tr>
<td>LD_LIBRARY_PATH</td>
<td><code>lib</code> folders of your requirements.</td>
</tr>
<tr>
<td>DYLD_LIBRARY_PATH</td>
<td><code>lib</code> folders of your requirements.</td>
</tr>
</tbody>
</table>

### 14.4.21 youcompleteme

Go to *Integrations/YouCompleteMe* to see the details of the YouCompleteMe generator.

### 14.4.22 txt

This is the reference page for `txt` generator. Go to *Integrations/Custom integrations / Use the text generator* to know how to use it.
File format

The generated `conanbuildinfo.txt` file is a generic config file with `[sections]` and values.

Package declared vars

For each requirement `conanbuildinfo.txt` file declares the following sections. `XXX` is the name of the require in lowercase. e.k “zlib” for `zlib/1.2.8@lasote/stable` requirement:

<table>
<thead>
<tr>
<th>SECTION</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>[include_dirs_XXX]</td>
<td>List with the include paths of the requirement</td>
</tr>
<tr>
<td>[libdirs_XXX]</td>
<td>List with library paths of the requirement</td>
</tr>
<tr>
<td>[bindirs_XXX]</td>
<td>List with binary directories of the requirement</td>
</tr>
<tr>
<td>[resdirs_XXX]</td>
<td>List with the resource directories of the requirement</td>
</tr>
<tr>
<td>[builddirs_XXX]</td>
<td>List with the build directories of the requirement</td>
</tr>
<tr>
<td>[libs_XXX]</td>
<td>List with library names of the requirement</td>
</tr>
<tr>
<td>[defines_XXX]</td>
<td>List with the defines of the requirement</td>
</tr>
<tr>
<td>[cflags_XXX]</td>
<td>List with C compilation flags</td>
</tr>
<tr>
<td>[sharedlinkflags_XXX]</td>
<td>List with shared libraries link flags</td>
</tr>
<tr>
<td>[exelinkflags_XXX]</td>
<td>List with executable link flags</td>
</tr>
<tr>
<td>[cppflags_XXX]</td>
<td>List with C++ compilation flags</td>
</tr>
<tr>
<td>[rootpath_XXX]</td>
<td>Root path of the package</td>
</tr>
</tbody>
</table>

Global declared vars

Conan also declares some global variables with the aggregated values of all our requirements. The values are ordered in the right order according to the dependency tree.

<table>
<thead>
<tr>
<th>SECTION</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>[include_dirs]</td>
<td>List with the aggregated include paths of the requirements</td>
</tr>
<tr>
<td>[libdirs]</td>
<td>List with aggregated library paths of the requirements</td>
</tr>
<tr>
<td>[bindirs]</td>
<td>List with aggregated binary directories of the requirements</td>
</tr>
<tr>
<td>[resdirs]</td>
<td>List with the aggregated resource directories of the requirements</td>
</tr>
<tr>
<td>[builddirs]</td>
<td>List with the aggregated build directories of the requirements</td>
</tr>
<tr>
<td>[libs]</td>
<td>List with aggregated library names of the requirements</td>
</tr>
<tr>
<td>[defines]</td>
<td>List with the aggregated defines of the requirements</td>
</tr>
<tr>
<td>[cflags]</td>
<td>List with aggregated C compilation flags</td>
</tr>
<tr>
<td>[sharedlinkflags]</td>
<td>List with aggregated shared libraries link flags</td>
</tr>
<tr>
<td>[exelinkflags]</td>
<td>List with aggregated executable link flags</td>
</tr>
<tr>
<td>[cppflags]</td>
<td>List with aggregated C++ compilation flags</td>
</tr>
</tbody>
</table>

14.4.23 json

**Warning:** Actual JSON may have more fields not documented here. Those fields may change in the future without previous warning.

A file named `conanbuildinfo.json` will be generated. It will contain the information about every dependency and the installed settings and options:
The generated `conanbuildinfo.json` file is a JSON file with the following keys:

**dependencies**

The dependencies is a list, with each item belonging to one dependency, and each one with the following keys:

- name
- version
- description
- rootpath
- sysroot
- include_paths, lib_paths, bin_paths, build_paths, res_paths
• **libs**

• **defines, cflags, cppflags, sharedlinkflags, exelinkflags**

• **configs** (only for multi config dependencies, see below)

Please note that the dependencies are ordered, it isn’t a map, order is relevant. Upstream dependencies, i.e. the ones that do not depend on other packages, will be first, and their direct dependencies after them, and so on.

The node **configs** will appear only for multi config recipes, it is holding a dictionary with the data related to each configuration:

```json
{
  ...
  "dependencies": [
    {
      "name": "Hello",
      "rootpath": "/private/var/folders/yq/14hmvxm96xd7qfg137_tnrbh0000gn/T/tmpkp91_.
    
    "...": "...",
    "configs": {
      "debug": {
        "libs": ["hello_d"]
      },
      "release": {
        "libs": ["hello"]
      }
    }
  },
  { "...": "..." }
}
```

**deps_env_info**

The environment variables defined by upstream dependencies

**deps_user_info**

The user variables defined by upstream dependencies

**settings**

The settings used during `conan install`

**options**

The options of each dependency
14.4.24 premake

Warning: This is an experimental feature subject to breaking changes in future releases.

This is the reference page for premake generator. Go to Integrations/premake if you want to learn how to integrate your project or recipes with premake.

Generates a file name conanbuildinfo.premake.lua that can be used for your premake builds (both premake 4 and premake 5 are supported).

The file contains:

- N groups of variables, one group per require, declaring the same individual values: include dirs, libs, bin dirs, defines, etc.
- One group of global variables with aggregated values for all requirements.
- Helper functions to setup the settings in your configuration.

Variables

Package declared variables

For each requirement conanbuildinfo.premake.lua file declares the following variables. XXX is the name of the require. e.g. “zlib” for zlib/1.2.11@lasote/stable requirement:

<table>
<thead>
<tr>
<th>NAME</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>conan_includedirs_XXX</td>
<td>Headers’s folders (default {CONAN_XXX_ROOT}/include)</td>
</tr>
<tr>
<td>conan_libdirs_XXX</td>
<td>Library folders (default {CONAN_XXX_ROOT}/lib)</td>
</tr>
<tr>
<td>conan_bindirs_XXX</td>
<td>Binary folders (default {CONAN_XXX_ROOT}/bin)</td>
</tr>
<tr>
<td>conan_libs_XXX</td>
<td>Library names to link</td>
</tr>
<tr>
<td>conan_defines_XXX</td>
<td>Compile definitions</td>
</tr>
<tr>
<td>conan_cxxflags_XXX</td>
<td>CXX flags</td>
</tr>
<tr>
<td>conan_cflags_XXX</td>
<td>C flags</td>
</tr>
<tr>
<td>conan_sharedlinkflags_XXX</td>
<td>Shared link flags</td>
</tr>
<tr>
<td>conan_exelinkflags_XXX</td>
<td>Executable link flags</td>
</tr>
<tr>
<td>conan_rootpath_XXX</td>
<td>Abs path to root package folder</td>
</tr>
</tbody>
</table>

Global declared variables

<table>
<thead>
<tr>
<th>NAME</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>conan_includedirs</td>
<td>Aggregated headers’s folders</td>
</tr>
<tr>
<td>conan_libdirs</td>
<td>Aggregated library folders</td>
</tr>
<tr>
<td>conan_bindirs</td>
<td>Aggregated binary folders</td>
</tr>
<tr>
<td>conan_libs</td>
<td>Aggregated library names to link</td>
</tr>
<tr>
<td>conan_defines</td>
<td>Aggregated compile definitions</td>
</tr>
<tr>
<td>conan_cxxflags</td>
<td>Aggregated CXX flags</td>
</tr>
<tr>
<td>conan_cflags</td>
<td>Aggregated C flags</td>
</tr>
<tr>
<td>conan_sharedlinkflags</td>
<td>Aggregated shared link flags</td>
</tr>
<tr>
<td>conan_exelinkflags</td>
<td>Aggregated executable link flags</td>
</tr>
</tbody>
</table>
**Functions**

**conan_basic_setup()**

Basic function to setup the settings into your configuration. Useful to reduce the logic in premake scripts and automate the conversion of settings:

```plaintext
function conan_basic_setup()
  configurations{conan_build_type}
  architecture(conan_arch)
  includedirs{conan_includedirs}
  libdirs{conan_libdirs}
  links{conan_libs}
  defines{conan_cppdefines}
  bindirs{conan_bindirs}
end
```

**14.4.25 make**

This is the reference page for make generator. Go to Integrations/make if you want to learn how to integrate your project or recipes with make.

This generators creates a file named `conanbuildinfo.mak` with information of dependencies in different variables that can be used for your make builds.

**Variables**

Variables per package. The `<PKG-NAME>` placeholder is filled with the name of the Conan package.

<table>
<thead>
<tr>
<th>NAME</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONAN_ROOT_&lt;PKG-NAME&gt;</td>
<td>Absolute path to root package folder</td>
</tr>
<tr>
<td>CONAN_SYSROOT_&lt;PKG-NAME&gt;</td>
<td>System root folder</td>
</tr>
<tr>
<td>CONAN_INCLUDE_DIRS_&lt;PKG-NAME&gt;</td>
<td>Headers folders</td>
</tr>
<tr>
<td>CONAN_LIB_DIRS_&lt;PKG-NAME&gt;</td>
<td>Library folders</td>
</tr>
<tr>
<td>CONAN_BIN_DIRS_&lt;PKG-NAME&gt;</td>
<td>Binary folders</td>
</tr>
<tr>
<td>CONAN_BUILD_DIRS_&lt;PKG-NAME&gt;</td>
<td>Build folders</td>
</tr>
<tr>
<td>CONAN_RES_DIRS_&lt;PKG-NAME&gt;</td>
<td>Resources folders</td>
</tr>
<tr>
<td>CONAN_LIBS_&lt;PKG-NAME&gt;</td>
<td>Library names to link with</td>
</tr>
<tr>
<td>CONAN_DEFINES_&lt;PKG-NAME&gt;</td>
<td>Library definitions</td>
</tr>
<tr>
<td>CONAN_CFLAGS_&lt;PKG-NAME&gt;</td>
<td>Options for the C compiler (-g, -s, -m64, -m32, -fPIC)</td>
</tr>
<tr>
<td>CONAN_CXXFLAGS_&lt;PKG-NAME&gt;</td>
<td>Options for the C++ compiler (-g, -s, -stdlib, -m64, -m32, -fPIC, -std)</td>
</tr>
<tr>
<td>CONAN_SHAREDLINKFLAGS_&lt;PKG-NAME&gt;</td>
<td>Library Shared linker flags</td>
</tr>
<tr>
<td>CONAN_EXELINK_FLAGS_&lt;PKG-NAME&gt;</td>
<td>Executable linker flags</td>
</tr>
</tbody>
</table>

Conan also declares some global variables with the aggregated values of all our requirements. The values are ordered in the right order according to the dependency tree.
<table>
<thead>
<tr>
<th>NAME</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONAN_ROOTPATH</td>
<td>Aggregated root folders</td>
</tr>
<tr>
<td>CONAN_SYSROOT</td>
<td>Aggregated system root folders</td>
</tr>
<tr>
<td>CONAN_INCLUDE_DIRS</td>
<td>Aggregated header folders</td>
</tr>
<tr>
<td>CONAN_LIB_DIRS</td>
<td>Aggregated library folders</td>
</tr>
<tr>
<td>CONAN_BIN_DIRS</td>
<td>Aggregated binary folders</td>
</tr>
<tr>
<td>CONAN_BUILD_DIRS</td>
<td>Aggregated build folders</td>
</tr>
<tr>
<td>CONAN_RES_DIRS</td>
<td>Aggregated resource folders</td>
</tr>
<tr>
<td>CONAN_LIBS</td>
<td>Aggregated library names to link with</td>
</tr>
<tr>
<td>CONAN_DEFINES</td>
<td>Aggregated library definitions</td>
</tr>
<tr>
<td>CONAN_CFLAGS</td>
<td>Aggregated options for the C compiler</td>
</tr>
<tr>
<td>CONAN_CXXFLAGS</td>
<td>Aggregated options for the C++ compiler</td>
</tr>
<tr>
<td>CONAN_SHARDELINKFLAGS</td>
<td>Aggregated Shared linker flags</td>
</tr>
<tr>
<td>CONAN_EXELINKFLAGS</td>
<td>Aggregated Executable linker flags</td>
</tr>
</tbody>
</table>

**Important:** Note that the mapping of the Conan variables to the Make ones is done taking the following rules and we suggest to use the variables indicated under the `Makefile` column to apply to a common naming:

<table>
<thead>
<tr>
<th>cpp_info</th>
<th>conanbuildinfo.mak</th>
<th>Makefile</th>
</tr>
</thead>
<tbody>
<tr>
<td>defines</td>
<td>CONAN_DEFINES</td>
<td>CPPFLAGS</td>
</tr>
<tr>
<td>includedirs</td>
<td>CONAN_INCLUDE_DIRS</td>
<td>CPPFLAGS</td>
</tr>
<tr>
<td>libdirs</td>
<td>CONAN_LIB_DIRS</td>
<td>LDFLAGS</td>
</tr>
<tr>
<td>libs</td>
<td>CONAN_LIBS</td>
<td>LDLIBS</td>
</tr>
<tr>
<td>cflags</td>
<td>CONAN_CFLAGS</td>
<td>CFLAGS</td>
</tr>
<tr>
<td>cxxflags</td>
<td>CONAN_CXXFLAGS</td>
<td>CXXFLAGS</td>
</tr>
</tbody>
</table>

### 14.4.26 deploy

The deploy generator makes a bulk copy of the packages folders of all dependencies in a graph. It can be used to deploy binaries from the local cache to the user space:

```
$ conan install OpenSSL/1.0.2r@conan/stable -g deploy
... Installing package: OpenSSL/1.0.2r@conan/stable
... Generator deploy created deploy_manifest.txt
```

Files form dependencies are deployed under a folder with the name of the dependency.

```
$ ls -R
OpenSSL/ conanbuildinfo.txt deploy_manifest.txt zlib/
./OpenSSL: LICENSE include/ lib/
./OpenSSL/include:
openssl/
./OpenSSL/include/openssl:
```

(continues on next page)
### 14.5 Profiles

Profiles allow users to set a complete configuration set for **settings**, **options**, **environment variables**, and **build requirements** in a file. They have this structure:

```plaintext
[settings]
setting=value

[options]
MyLib:shared=True
```

---

The generated `deploy_manifest.txt` file is a manifest file with a list of all the files deployed and hash of the contents for each of them.

**Tip:** You can use the parameter `--install-folder` in the `conan install` to output the contents of the packages to a specific folder.

**Important:** If none of these generators fit your needs, you can create your own custom_generator.
Profile can be created with new option in `conan profile`. And then edit it later.

```bash
$ conan profile new mynewprofile --detect
```

Profile files can be used with `-pr/--profile` option in `conan install` and `conan create` commands.

```bash
$ conan create . demo/testing -pr=myprofile
```

Profiles can be located in different folders. For example, the default `<userhome>/.conan/profiles`, and be referenced by absolute or relative path:

```bash
$ conan install . --profile /abs/path/to/profile  # abs path
$ conan install . --profile ./relpath/to/profile  # resolved to current dir
$ conan install . --profile profile  # resolved to user/.conan/profiles/profile
```

Listing existing profiles in the `profiles` folder can be done like this:

```bash
$ conan profile list
default
myprofile1
myprofile2
...
```

You can also show profile’s content:

```bash
$ conan profile show myprofile1
 Configuration for profile myprofile1:

[settings]
os=Windows
arch=x86_64
compiler=Visual Studio
compiler.version=15
build_type=Release
[options]
[build_requires]
[env]

Use `$PROFILE_DIR` in your profile and it will be replaced with the absolute path to the profile file. It is useful to declare relative folders:

```bash
[env]
PYTHONPATH=$PROFILE_DIR/my_python_tools
```

**Tip:** You can manage your profiles and share them using `conan config install`. 378 Chapter 14. Reference
14.5.1 Package settings and env vars

Profiles also support package settings and package environment variables definition, so you can override some settings or environment variables for some specific package:

Listing 9: `.conan/profiles/zlib_with_clang`

```bash
[settings]
zlib:compiler=clang
zlib:compiler.version=3.5
zlib:compiler.libcxx=libstdc++11
compiler=gcc
compiler.version=4.9
compiler.libcxx=libstdc++11

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

Your build tool will locate **clang** compiler only for the **zlib** package and **gcc** (default one) for the rest of your dependency tree.

**Note:** If you want to override existing system environment variables, you should use the key=value syntax. If you need to pre-pend to the system environment variables you should use the syntax key=[value] or key=[value1, value2, ...]. A typical example is the PATH environment variable, when you want to add paths to the existing system PATH, not override it, you would use:

```bash
[env]
PATH=/[some/path/to/my/tool]
```

14.5.2 Profile composition

You can specify multiple profiles in the command line. The applied configuration will be the composition of all the profiles applied in the order they are specified.

If, for example, you want to apply a **build require**, like a cmake installer to your dependency tree, it won’t be very practical adding the **cmake** installer reference, e.g cmake_installer/3.9.0@conan/stable to all your profiles where you could need to inject cmake as a build require.

You can specify both profiles instead:

Listing 10: `.conan/profiles/cmake_39`

```bash
[build_requires]
cmake_installer/3.9.0@conan/stable
```

$ conan install . --profile clang --profile cmake_39

14.5.3 Profile includes

You can include other profiles using the include() statement. The path can be relative to the current profile, absolute, or a profile name from the default profile location in the local cache.
The **include()** statement has to be at the top of the profile file:

### Listing 11: gcc_49

```ini
[settings]
compiler=gcc
compiler.version=4.9
compiler.libcxx=libstdc++11
```

### Listing 12: myprofile

```ini
include(gcc_49)

[settings]
zlib:compiler=clang
zlib:compiler.version=3.5
zlib:compiler.libcxx=libstdc++11

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

### 14.5.4 Variable declaration

In a profile you can declare variables that will be replaced automatically by Conan before the profile is applied. The variables have to be declared at the top of the file, after the **include()** statements.

### Listing 13: myprofile

```ini
include(gcc_49)
CLANG=/usr/bin/clang

[settings]
zlib:compiler=clang
zlib:compiler.version=3.5
zlib:compiler.libcxx=libstdc++11

[env]
zlib:CC=$CLANG/clang
zlib:CXX=$CLANG/clang++
```

The variables will be inherited too, so you can declare variables in a profile and then include the profile in a different one, all the variables will be available:

### Listing 14: gcc_49

```ini
GCC_PATH=/my/custom/toolchain/path/

[settings]
compiler=gcc
compiler.version=4.9
compiler.libcxx=libstdc++11
```
14.5.5 Examples

If you are working with Linux and you usually work with gcc compiler, but you have installed clang compiler and want to install some package for clang compiler, you could do:

- Create a .conan/profiles/clang file:

```python
[settings]
compiler=clang
compiler.version=3.5
compiler.libcxx=libstdc++11
[env]
CC=$GCC_PATH/gcc
CXX=$GCC_PATH/g++
```

- Execute an install command passing the --profile or -pr parameter:

```
$ conan install . --profile clang
```

Without profiles you would have needed to set CC and CXX variables in the environment to point to your clang compiler and use -s parameters to specify the settings:

```
$ export CC=/usr/bin/clang
$ export CXX=/usr/bin/clang++
$ conan install -s compiler=clang -s compiler.version=3.5 -s compiler.libcxx=libstdc++11
```

A profile can also be used in conan create and conan info:

```
$ conan create . demo/testing --profile clang
```

See also:

- Check the section Build requirements to read more about its usage in a profile.
- Check conan profile and profiles/default for full reference.
- Related section: Cross-building.
14.6 Build helpers

Build helpers are Python wrappers of a build tool that help with the conversion of the Conan settings to the build system’s ones. They assist users with the compilation of libraries and applications in the `build()` method of a recipe.

Contents:

14.6.1 CMake

The *CMake* class helps us to invoke *cmake* command with the generator, flags and definitions, reflecting the specified Conan settings.

There are two ways to invoke your cmake tools:

- Using the helper attributes `cmake.command_line` and `cmake.build_config`:

```python
from conans import ConanFile, CMake

class ExampleConan(ConanFile):
    ...

    def build(self):
        cmake = CMake(self)
        self.run('cmake "%s" %s % (self.source_folder, cmake.command_line))
        self.run('cmake --build . %s cmake.build_config)
        self.run('cmake --build . --target install')
```

- Using the helper methods:

```python
from conans import ConanFile, CMake

class ExampleConan(ConanFile):
    ...

    def build(self):
        cmake = CMake(self)
        # same as cmake.configure(source_folder=self.source_folder, build_folder=self.
        # build_folder)
        cmake.configure()
        cmake.build()
        cmake.test() # Build the "RUN_TESTS" or "test" target
        # Build the "install" target, defining CMAKE_INSTALL_PREFIX to self.package_  
        # folder
        cmake.install()
```

Constructor

```python
class CMake(object):
    
    def __init__(self, conanfile, generator=None, cmake_system_name=True,
                 parallel=True, build_type=None, toolset=None, make_program=None,
                 set_cmake_flags=False, msbuild_verbosity='minimal', cmake_  
                 program=None,
                 generator_platform=None)
```
Parameters:

- **conanfile** (Required): Conanfile object. Usually `self` in a `conanfile.py`  
- **generator** (Optional, Defaulted to None): Specify a custom generator instead of autodetect it. e.g., “MinGW Makefiles”  
- **cmake_system_name** (Optional, Defaulted to True): Specify a custom value for `CMAKE_SYSTEM_NAME` instead of autodetect it.  
- **parallel** (Optional, Defaulted to True): If True, will append the `-jN` attribute for parallel building being N the `cpu_count()`. Also applies to parallel test execution (by defining `CTEST_PARALLEL_LEVEL` environment variable).  
- **build_type** (Optional, Defaulted to None): Force the build type instead of taking the value from the settings. Note this will also make the `CMAKE_BUILD_TYPE` to be declared for multi configuration generators.  
- **toolset** (Optional, Defaulted to None): Specify a toolset for Visual Studio.  
- **make_program** (Optional, Defaulted to None): Indicate path to `make`.  
- **set_cmake_flags** (Optional, Defaulted to None): Whether or not to set CMake flags like `CMAKE_CXX_FLAGS`, `CMAKE_C_FLAGS`, etc.  
- **msbuild_verbosity** (Optional, Defaulted to minimal): verbosity level for MSBuild (in case of Visual Studio generator). Set this parameter to None to avoid using it in the command line.  
- **cmake_program** (Optional, Defaulted to None): Path to the custom `make` executable.  
- **generator_platform** (Optional, Defaulted to None): Generator platform name or none to autodetect (-A cmake option).  

Attributes

**generator**

Specifies a custom CMake generator to use, see also `cmake-generators documentation`.  

**generator_platform**

Specifies a custom CMake generator platform to use, see also `CMAKE_GENERATOR_PLATFORM documentation`.  

**verbose**

Defaulted to: False

Set it to `True` or `False` to automatically set the definition `CMAKE_VERBOSE_MAKEFILE`.  

```python
from conans import ConanFile, CMake

class ExampleConan(ConanFile):

    def build(self):
        cmake = CMake(self)
        cmake.verbose = True
```

(continues on next page)
build_folder (Read only)

Build folder where the configure() and build() methods will be called.

build_type [Deprecated]

Build type can be forced with this variable instead of taking it from the settings.

flags (Read only)

Flag conversion of definitions to be used in the command line invocation (-D).

is_multi_configuration (Read only)

Indicates whether the generator selected allows builds with multi configuration: Release, Debug... Multi configuration generators are Visual Studio and Xcode ones.

command_line (Read only)

Arguments and flags calculated by the build helper that will be applied. It indicates the generator, the Conan definitions and the flags converted from the specified Conan settings. For example:

```
-G "Unix Makefiles" -DCMAKE_BUILD_TYPE=Release ... -DCONAN_C_FLAGS=-m64 -Wno-dev
```

build_config (Read only)

Value for --config option for Multi-configuration IDEs. This flag will only be set if the generator is_multi_configuration and build_type was not forced in constructor class.

An example of the value of this property could be:

```
--config Release
```

parallel

Defaulted to: True

Run CMake process in parallel for compilation, installation and testing. This is translated into the proper command line argument: For Unix Makefiles it is -jX and for Visual Studio it is /m:X.

However, the parallel executing can be changed for testing like this:

```
In the case of `cmake.test()` this flag sets the `CTEST_PARALLEL_LEVEL` variable to the according value in `tools.cpu_count()`.

definitions

The CMake helper will automatically append some definitions based on your settings:
<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMAKE_BUILD_TYPE</td>
<td>Debug, Release... from self.settings.build_type or build_type attribute only if is_multi_configuration</td>
</tr>
<tr>
<td>CMAKE_OSX_ARCHITECTURES</td>
<td>i386 if architecture is x86 in an OSX system</td>
</tr>
<tr>
<td>BUILD_SHARED_LIBS</td>
<td>Conan internal variable to check the compiler</td>
</tr>
<tr>
<td>CONAN_COMPILER</td>
<td>Only if your recipe has a shared option</td>
</tr>
<tr>
<td>CMAKE_SYSTEM_NAME</td>
<td>Set to self.settings.os value if cross-building is detected</td>
</tr>
<tr>
<td>CMAKE_SYSTEM_VERSION</td>
<td>Set to self.settings.os_version value if cross-building is detected</td>
</tr>
<tr>
<td>CMAKE_ANDROID_ARCH_ABI</td>
<td>Set to a suitable value if cross-building to an Android is detected</td>
</tr>
<tr>
<td>CONAN_LIBCXX</td>
<td>Set to self.settings.compiler.libcxx value</td>
</tr>
<tr>
<td>CONAN_CMAKE_SYSTEM_PROCESSOR</td>
<td>Definition set only if same environment variable is declared by user</td>
</tr>
<tr>
<td>CO-NAN_CMAKE_FIND_ROOT_PATH</td>
<td>Definition set only if same environment variable is declared by user</td>
</tr>
<tr>
<td>CO-NAN_CMAKE_FIND_ROOT_PATH_MODE_PROGRAM</td>
<td>Definition set only if same environment variable is declared by user for MODE_PROGRAM</td>
</tr>
<tr>
<td>CO-NAN_CMAKE_FIND_ROOT_PATH_MODE_LIBRARY</td>
<td>Definition set only if same environment variable is declared by user for MODE_LIBRARY</td>
</tr>
<tr>
<td>CO-NAN_CMAKE_FIND_ROOT_PATH_MODE_INCLUDE</td>
<td>Definition set only if same environment variable is declared by user for MODE_INCLUDE</td>
</tr>
<tr>
<td>CO-NAN_CMAKE_POSITION_INDEPENDENT_CODE</td>
<td>Set when fPIC option exists and True or fPIC exists and False but shared option exists and True</td>
</tr>
<tr>
<td>CONAN_SHARED_LINKER_FLAGS</td>
<td>Set to -m32 or -m64 values based on the architecture</td>
</tr>
<tr>
<td>CONAN_C_FLAGS</td>
<td>Set to -m32 or -m64 values based on the architecture and /MP for MSVS</td>
</tr>
<tr>
<td>CONAN_CXX_FLAGS</td>
<td>Set to -m32 or -m64 values based on the architecture and /MP for MSVS</td>
</tr>
<tr>
<td>CONAN_LINK_RUNTIME</td>
<td>Set to the runtime value from self.settings.compiler.runtime for MSVS</td>
</tr>
<tr>
<td>CO-NAN_CMAKE_CXX_STANDARD</td>
<td>Set to the self.settings.compiler.cppstd value (or self.settings.cppstd for backward compatibility)</td>
</tr>
<tr>
<td>CO-NAN_CMAKE_CXX_EXTENSIONS</td>
<td>Set to ON or OFF value when GNU extensions for the given C++ standard are enabled</td>
</tr>
<tr>
<td>CONAN_STD_CXX_FLAG</td>
<td>Set to the flag corresponding to the C++ standard defined in self.settings.compiler.cppstd. Used for CMake &lt; 3.1)</td>
</tr>
<tr>
<td>CMAKE_EXPORT_NO_PACKAGE_REGISTRY</td>
<td>Admistically default to disable the package registry</td>
</tr>
<tr>
<td>CONAN_IN_LOCAL_CACHE</td>
<td>ON if the build runs in local cache, OFF if running in a user folder</td>
</tr>
<tr>
<td>CONAN_EXPORTED</td>
<td>Defined when CMake is called using Conan CMake helper</td>
</tr>
<tr>
<td>ANDROID_ABI</td>
<td>Just alias for CMAKE_ANDROID_ARCH_ABI</td>
</tr>
<tr>
<td>ANDROID_NDK</td>
<td>Defined when one of ANDROID_NDK_ROOT or ANDROID_NDK_HOME environment variables presented</td>
</tr>
</tbody>
</table>

There are some definitions set to be used later on the the install() step too:
<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMAKE_INSTALL_PREFIX</td>
<td>Set to <code>conanfile.package_folder</code> value.</td>
</tr>
<tr>
<td>CMAKE_INSTALL_BINDIR</td>
<td>Set to <code>bin</code> inside the package folder.</td>
</tr>
<tr>
<td>CMAKE_INSTALL_SBINDIR</td>
<td>Set to <code>bin</code> inside the package folder.</td>
</tr>
<tr>
<td>CMAKE_INSTALL_LIBEXECDIR</td>
<td>Set to <code>bin</code> inside the package folder.</td>
</tr>
<tr>
<td>CMAKE_INSTALL_LIBDIR</td>
<td>Set to <code>lib</code> inside the package folder.</td>
</tr>
<tr>
<td>CMAKE_INSTALL_INCLUDEDIR</td>
<td>Set to <code>include</code> inside the package folder.</td>
</tr>
<tr>
<td>CMAKE_INSTALL_OLDINCLUDEDIR</td>
<td>Set to <code>include</code> inside the package folder.</td>
</tr>
<tr>
<td>CMAKE_INSTALL_DATAROOTDIR</td>
<td>Set to <code>share</code> inside the package folder.</td>
</tr>
</tbody>
</table>

But you can change the automatic definitions after the `CMake()` object creation using the `definitions` property or even add your own ones:

```python
from conans import ConanFile, CMake

class ExampleConan(ConanFile):
    ...

    def build(self):
        cmake = CMake(self)
        cmake.definitions["CMAKE_SYSTEM_NAME"] = "Generic"
        cmake.definitions["MY_CUSTOM_DEFINITION"] = True
        cmake.configure()
        cmake.build()
        cmake.install()  # Build --target=install
```

Note that definitions changed after the `configure()` call will **not** take effect later on the `build()`, `test()` or `install()` ones.

**Methods**

**configure()**

```python
def configure(self, args=None, defs=None, source_dir=None, build_dir=None,
source_folder=None, build_folder=None, cache_build_folder=None,
pkg_config_paths=None)
```

Configures `CMake` project with the given parameters.

**Parameters:**

- `args` (Optional, Defaulted to `None`): A list of additional arguments to be passed to the `cmake` command. Each argument will be escaped according to the current shell. No extra arguments will be added if `args=None`
- `defs` (Optional, Defaulted to `None`): A dict that will be converted to a list of CMake command line variable definitions of the form `-DKEY=VALUE`. Each value will be escaped according to the current shell and can be either `str`, `bool` or of numeric type
- `source_dir` (Optional, Defaulted to `None`): [DEPRECATED] Use `source_folder` instead. CMake’s source directory where `CMakeLists.txt` is located. The default value is the `build` folder if `None` is specified (or the `source` folder if `no_copy_source` is specified). Relative paths are allowed and will be relative to `build_folder`.

14.6. Build helpers
build_dir (Optional, Defaulted to None): [DEPRECATED] Use build_folder instead. CMake’s output directory. The default value is the package build root folder if None is specified. The CMake object will store build_folder internally for subsequent calls to build().

source_folder: CMake’s source directory where CMakeLists.txt is located. The default value is the self.source_folder. Relative paths are allowed and will be relative to self.source_folder.

build_folder: CMake’s output directory. The default value is the self.build_folder if None is specified. The CMake object will store build_folder internally for subsequent calls to build().

cache_build_folder (Optional, Defaulted to None): Use the given subfolder as build folder when building the package in the local cache. This argument doesn’t have effect when the package is being built in user folder with conan build but overrides build_folder when working in the local cache. See self.in_local_cache.

pkg_config_paths (Optional, Defaulted to None): Specify folders (in a list) of relative paths to the install folder or absolute ones where to find *.pc files (by using the env var PKG_CONFIG_PATH). If None is specified but the conanfile is using the pkg_config generator, the self.install_folder will be added to the PKG_CONFIG_PATH in order to locate the pc files of the requirements of the conanfile.

build()

```python
def build(self, args=None, build_dir=None, target=None)
```

Builds CMake project with the given parameters.

Parameters:
- **args** (Optional, Defaulted to None): A list of additional arguments to be passed to the cmake command. Each argument will be escaped according to the current shell. No extra arguments will be added if args=None
- **build_dir** (Optional, Defaulted to None): CMake’s output directory. If None is specified the build_dir from configure() will be used.
- **target** (Optional, Defaulted to None): Specifies the target to execute. The default all target will be built if None is specified. "install" can be used to relocate files to aid packaging.

test()

```python
def test(args=None, build_dir=None, target=None, output_on_failure=False)
```

Build CMake test target (could be RUN_TESTS in multi-config projects or test in single-config projects), which usually means building and running unit tests

Parameters:
- **args** (Optional, Defaulted to None): A list of additional arguments to be passed to the cmake command. Each argument will be escaped according to the current shell. No extra arguments will be added if args=None.
- **build_dir** (Optional, Defaulted to None): CMake’s output directory. If None is specified the build_folder from configure() will be used.
- **target** (Optional, default to None). Alternative target name for running the tests. If not defined RUN_TESTS or test will be used.
• **output_on_failure** (Optional, default to `False`). Enables ctest to show output of failed tests by defining `CTEST_OUTPUT_ON_FAILURE` environment variable (same effect as `ctest --output-on-failure`).

### install()

```python
def install(args=None, build_dir=None)
```

Installs CMake project with the given parameters.

**Parameters:**

- **args** (Optional, Defaulted to `None`): A list of additional arguments to be passed to the `cmake` command. Each argument will be escaped according to the current shell. No extra arguments will be added if `args=None`.
- **build_dir** (Optional, Defaulted to `None`): CMake’s output directory. If `None` is specified the `build_folder` from `configure()` will be used.

### patch_config_paths() [EXPERIMENTAL]

```python
def patch_config_paths()
```

**Warning:** This is an experimental feature subject to breaking changes in future releases.

This method changes references to the absolute path of the installed package in exported CMake config files to the appropriate Conan variable. Method also changes references to other packages installation paths in export CMake config files to Conan variable with their installation roots. This makes most CMake config files portable.

For example, if a package foo installs a file called `fooConfig.cmake` to be used by cmake’s `find_package()` method, normally this file will contain absolute paths to the installed package folder, for example it will contain a line such as:

```bash
SET(Foo_INSTALL_DIR /home/developer/.conan/data/Foo/1.0.0/...)
```

This will cause cmake’s `find_package()` method to fail when someone else installs the package via Conan. This function will replace such paths to:

```bash
SET(Foo_INSTALL_DIR ${CONAN_FOO_ROOT})
```

Which is a variable that is set by `conanbuildinfo.cmake`, so that `find_package()` now correctly works on this Conan package.

For dependent packages method replaces lines with references to dependencies installation paths such as:

```bash
SET_TARGET_PROPERTIES(foo PROPERTIES INTERFACE_INCLUDEDIRECTORIES "~/home/developer/.conan/data/Bar/1.0.0/user/channel/id/include")
```

to following lines:

```bash
SET_TARGET_PROPERTIES(foo PROPERTIES INTERFACE_INCLUDEDIRECTORIES "${CONAN_BAR_ROOT}/include")
```
If the `install()` method of the CMake object in the conanfile is used, this function should be called after that invocation. For example:

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

### get_version()

```python
@staticmethod
def get_version():
```

Returns the CMake version in a `conans.model.Version` object as it is evaluated by the command line. Will raise if cannot resolve it to valid version.

### Environment variables

There are some environment variables that will also affect the `CMake()` helper class. Check them in the `CMAKE RELATED VARIABLES` section.

### Example

The following example of `conanfile.py` shows you how to manage a project with conan and CMake.

```python
from conans import ConanFile, CMake

class SomePackage(ConanFile):
    name = "SomePackage"
    version = "1.0.0"
    settings = "os", "compiler", "build_type", "arch"
    generators = "cmake"

    def configure_cmake(self):
        cmake = CMake(self)

        # put definitions here so that they are re-used in cmake between
        # build() and package()
        cmake.defintions["SOME_DEFINITION_NAME"] = "On"

        cmake.configure()
        return cmake

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

        # run unit tests after the build
        cmake.test()

        # run custom make command
```

(continues on next page)
self.run("make -j3 check")

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

Default used generators

When a compiler or its version is not detected, the CMake helper uses a default generator based on the platform operating system. For Unix systems it generates Unix Makefiles. For Windows there is no default generator, it will be detected by CMake automatically.

14.6.2 AutoToolsBuildEnvironment (configure/make)

If you are using configure/make you can use AutoToolsBuildEnvironment helper. This helper sets LIBS, LDFLAGS, CFLAGS, CXXFLAGS and CPPFLAGS environment variables based on your requirements.

from conans import ConanFile, AutoToolsBuildEnvironment

class ExampleConan(ConanFile):
    settings = "os", "compiler", "build_type", "arch"
    requires = "Poco/1.9.0@pocoproject/stable"
    default_options = {"Poco:shared": True, "OpenSSL:shared": True}

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

    def build(self):
        autotools = AutoToolsBuildEnvironment(self)
        autotools.configure()
        autotools.make()

It also works using the environment_append context manager applied to your configure and make commands, calling configure and make manually:

from conans import ConanFile, AutoToolsBuildEnvironment, tools

class ExampleConan(ConanFile):
    ...

    def build(self):
        env_build = AutoToolsBuildEnvironment(self)
        with tools.environment_append(env_build.vars):
            self.run("./configure")
            self.run("make")

You can change some variables like fpic, libs, include_paths and defines before accessing the vars to override an automatic value or add new values:
...  

```python
def build(self):
    env_build = AutoToolsBuildEnvironment(self)
    env_build.fpic = True
    env_build.libs.append("pthread")
    env_build.defines.append("NEW_DEFINE=23")
    env_build.configure()
    env_build.make()
```

You can use it also with MSYS2/MinGW subsystems installed by setting the `win_bash` parameter in the constructor. It will run the `configure` and `make` commands inside a `bash` that has to be in the path or declared in `CONAN_BASH_PATH`:

```
from conans import ConanFile, AutoToolsBuildEnvironment, tools

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

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

    def build(self):
        env_build = AutoToolsBuildEnvironment(self, win_bash=tools.os_info.is_windows)
        env_build.configure()
        env_build.make()
```

### Constructor

```
class AutoToolsBuildEnvironment(object):
    def __init__(self, conanfile, win_bash=False)
```

**Parameters:**

- `conanfile` (Required): ConanFile object. Usually `self` in a conanfile.py
- `win_bash` (Optional, Defaulted to `False`): When True, it will run the `configure/make` commands inside a `bash`.

### Attributes

You can adjust the automatically filled values modifying the attributes like this:

```
from conans import ConanFile, AutoToolsBuildEnvironment

class ExampleConan(ConanFile):
    ...

    def build(self):
        autotools = AutoToolsBuildEnvironment(self)
        autotools.fpic = True
        autotools.libs.append("pthread")
```

(continues on next page)
autotools.defines.append("NEW_DEFINE=23")
autotools.configure()
autotools.make()

**fpic**

Defaulted to: **True** if **fPIC** option exists and **True** or when **fPIC** exists and **False** but option **shared** exists and **True**. Otherwise **None**.  
Set it to **True** if you want to append the `-fPIC` flag.

**libs**

List with library names of the requirements (-l in LIBS).

**include_paths**

List with the include paths of the requires (-I in CPPFLAGS).

**library_paths**

List with library paths of the requirements (-L in LDFLAGS).

**defines**

List with variables that will be defined with `-D` in CPPFLAGS.

**flags**

List with compilation flags (CFLAGS and CXXFLAGS).

**cxx_flags**

List with only C++ compilation flags (CXXFLAGS).

**link_flags**

List with linker flags
Properties

vars

Environment variables CPPFLAGS, CXXFLAGS, CFLAGS, LDFLAGS, LIBS generated by the build helper to use them in the configure, make and install steps. This variables are generated dynamically with the values of the attributes and can also be modified to be used in the following configure, make or install steps:

```python
def build():
    autotools = AutoToolsBuildEnvironment()
    autotools.fpic = True
    env_build_vars = autotools.vars
    env_build_vars['RCFLAGS'] = '-O COFF'
    autotools.configure(vars=env_build_vars)
    autotools.make(vars=env_build_vars)
    autotools.install(vars=env_build_vars)
```

vars_dict

Same behavior as vars but this property returns each variable CPPFLAGS, CXXFLAGS, CFLAGS, LDFLAGS, LIBS as dictionaries.

Methods

configure()

```python
def configure(self, configure_dir=None, args=None, build=None, host=None, target=None,
            pkg_config_paths=None, vars=None)
```

Configures Autotools project with the given parameters.

**Important:** This method sets by default the --prefix argument to self.package_folder whenever --prefix is not provided in the args parameter during the configure step.

There are other flags set automatically to fix the install directories by default:

- --bindir, --sbindir and --libexecdir set to *bin* folder.
- --libdir set to *lib* folder.
- --includedir, --oldincludedir set to *include* folder.
- --datarootdir set to *share* folder.

These flags will be set on demand, so only the available options in the *./configure* are actually set. They can also be totally skipped using use_default_install_dirs=False as described in the section below.

**Warning:** Since Conan 1.8 this build helper sets the output library directory via --libdir automatically to ${prefix}/lib. This means that if you are using the install() method to package with AutoTools, library artifacts will be stored in the *lib* directory unless indicated explicitly by the user.

This change was introduced in order to fix issues detected in some Linux distributions where libraries were being installed to the *lib64* folder (instead of *lib*) when rebuilding a package from sources. In those cases, if
package_info() was declaring self.cpp_info.libdirs as lib, the consumption of the package was broken.

This was considered a bug in the build helper, as it should be as much deterministic as possible when building the same package for the same settings and generally for any other user input.

If you were already modeling the lib64 folder in your recipe, make sure you use lib for self.cpp_info.libdirs or inject the argument in the Autotools' configure() method:

```python
atools = AutoToolsBuildEnvironment()
atools.configure(args=["--libdir=${prefix}/lib64"])
atools.install()
```

You can also skip its default value using the parameter `use_default_install_dirs=False`.

**Parameters:**

- **configure_dir** (Optional, Defaulted to `None`): Directory where the configure script is. If `None`, it will use the current directory.

- **args** (Optional, Defaulted to `None`): A list of additional arguments to be passed to the configure script. Each argument will be escaped according to the current shell. `--prefix` and `--libdir`, will be adjusted automatically if not indicated specifically.

- **build** (Optional, Defaulted to `None`): To specify a value for the parameter `--build`. If `None` it will try to detect the value if cross-building is detected according to the settings. If `False`, it will not use this argument at all.

- **host** (Optional, Defaulted to `None`): To specify a value for the parameter `--host`. If `None` it will try to detect the value if cross-building is detected according to the settings. If `False`, it will not use this argument at all.

- **target** (Optional, Defaulted to `None`): To specify a value for the parameter `--target`. If `None` it will try to detect the value if cross-building is detected according to the settings. If `False`, it will not use this argument at all.

- **pkg_config_paths** (Optional, Defaulted to `None`): Specify folders (in a list) of relative paths to the install folder or absolute ones where to find *.pc files (by using the env var `PKG_CONFIG_PATH`). If `None` is specified but the conanfile is using the pkg_config generator, the `self.install_folder` will be added to the `PKG_CONFIG_PATH` in order to locate the pc files of the requirements of the conanfile.

- **vars** (Optional, Defaulted to `None`): Overrides custom environment variables in the configure step.

- **use_default_install_dirs** (Optional, Defaulted to `True`): Use or not the defaulted installation dirs such as `--libdir`, `--bindir`...

**make()**

```python
def make(self, args="", make_program=None, target=None, vars=None)
```

Builds Autotools project with the given parameters.

**Parameters:**

- **args** (Optional, Defaulted to ""): A list of additional arguments to be passed to the make command. Each argument will be escaped accordingly to the current shell. No extra arguments will be added if `args=""`.

- **make_program** (Optional, Defaulted to `None`): Allows to specify a different make executable, e.g., mingw32-make. The environment variable `CONAN_MAKE_PROGRAM` can be used too.
• **target** (Optional, Defaulted to **None**): Choose which target to build. This allows building of e.g., docs, shared libraries or install for some AutoTools projects.

• **vars** (Optional, Defaulted to **None**): Overrides custom environment variables in the make step.

### install()

```python
def install(self, args='', make_program=None, vars=None)
```

Performs the install step of autotools calling `make(target="install")`.

**Parameters:**

• **args** (Optional, Defaulted to " "): A list of additional arguments to be passed to the `make` command. Each argument will be escaped accordingly to the current shell. No extra arguments will be added if `args=""`.

• **make_program** (Optional, Defaulted to **None**): Allows to specify a different `make` executable, e.g., `mingw32-make`. The environment variable `CONAN_MAKE_PROGRAM` can be used too.

• **vars** (Optional, Defaulted to **None**): Overrides custom environment variables in the install step.

### Environment variables

The following environment variables will also affect the `AutoToolsBuildEnvironment` helper class.

<table>
<thead>
<tr>
<th>NAME</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIBS</td>
<td>Library names to link</td>
</tr>
<tr>
<td>LDFLAGS</td>
<td>Link flags, (-L, -m64, -m32)</td>
</tr>
<tr>
<td>CFLAGS</td>
<td>Options for the C compiler (-g, -s, -m64, -m32, -fPIC)</td>
</tr>
<tr>
<td>CXXFLAGS</td>
<td>Options for the C++ compiler (-g, -s, -stdlib, -m64, -m32, -fPIC, -std)</td>
</tr>
<tr>
<td>CPPFLAGS</td>
<td>Preprocessor definitions (-D, -I)</td>
</tr>
</tbody>
</table>

See also:

• [Reference/Tools/environment_append](#reference-tools/environment_append)

### 14.6.3 MSBuild

Calls Visual Studio **MSBuild** command to build a `.sln` project:

```python
from conans import ConanFile, MSBuild

class ExampleConan(ConanFile):
    ...

    def build(self):
        msbuild = MSBuild(self)
        msbuild.build("MyProject.sln")
```

Internally the **MSBuild** build helper uses **VisualStudioBuildEnvironment** to adjust the LIB and CL environment variables with all the information from the requirements: include directories, library names, flags etc. and then calls **MSBuild**.

• **VisualStudioBuildEnvironment** to adjust the LIB and CL environment variables with all the information from the requirements: include directories, library names, flags etc.
You can adjust all the information from the requirements accessing to the `build_env` that it is a `VisualStudioBuildEnvironment` object:

```python
from conans import ConanFile, MSBuild

class ExampleConan(ConanFile):
    ...
    def build(self):
        msbuild = MSBuild(self)
        msbuild.build_env.include_paths.append("mycustom/directory/to/headers")
        msbuild.build_env.lib_paths.append("mycustom/directory/to/libs")
        msbuild.build_env.link_flags = []
        msbuild.build("MyProject.sln")
```

To inject the flags corresponding to the `compiler.runtime`, `build_type` and `compiler.cppstd` settings, this build helper also generates a properties file (in the build folder) that is passed to :command:`MSBuild` with:

:command:`/p:ForceImportBeforeCppTargets="conan_build.props"`

### Constructor

```python
class MSBuild(object):
    def __init__(self, conanfile)
```

#### Parameters:

- **conanfile** (Required): ConanFile object. Usually `self` in a `conanfile.py`.

### Attributes

**build_env**

A `VisualStudioBuildEnvironment` object with the needed environment variables.

### Methods

**build()**

```python
def build(self, project_file, targets=None, upgrade_project=True, build_type=None, ...
    --arch=None,
    parallel=True, force_vcvars=False, toolset=None, use_
    --env=True,
    vcvars_ver=None, winsdk_version=None, properties=None, output_binary_
    --log=None,
    property_file_name=None, verbosity=None, definitions=None)
```

Builds Visual Studio project with the given parameters.

#### Parameters:

- **project_file** (Required): Path to the `.sln` file.
• **targets** (Optional, Defaulted to `None`): Sets `/target` flag to the specified list of targets to build.

• **upgrade_project** (Optional, Defaulted to `True`): Will call `devenv /upgrade` to upgrade the solution to your current Visual Studio.

• **build_type** (Optional, Defaulted to `None`): Sets `/p:Configuration` flag to the specified value. It will override the value from `settings.build_type`.

• **arch** (Optional, Defaulted to `None`): Sets `/p:Platform` flag to the specified value. It will override the value from `settings.arch`. This value (or the `settings.arch` one if not overridden) will be used as the key for the `msvc_arch` dictionary that returns the final string used for the `/p:Platform` flag (see `platforms` argument documentation below).

• **parallel** (Optional, Defaulted to `True`): Will use the configured number of cores in the `conan.conf` file or `tools.cpu_count()`:
  
  – **In the solution**: Building the solution with the projects in parallel. (`/m:` parameter).
  

• **force_vcvars** (Optional, Defaulted to `False`): Will ignore if the environment is already set for a different Visual Studio version.

• **toolset** (Optional, Defaulted to `None`): Sets `/p:PlatformToolset` to the specified toolset. When `None` it will apply the setting `compiler.toolset` if specified. When `False` it will skip adjusting the `/p:PlatformToolset`.

• **platforms** (Optional, Defaulted to `None`): This dictionary will update the default one (see `msvc_arch` below) and will be used to get the mapping of architectures to platforms from the Conan naming to another one. It is useful for Visual Studio solutions that have a different naming in architectures. Example: `platforms={"x86":"Win32"}` (Visual solution uses “Win32” instead of “x86”).

```python
msvc_arch = {'x86': 'x86',
              'x86_64': 'x64',
              'armv7': 'ARM',
              'armv8': 'ARM64'}
```

• **use_env** (Optional, Defaulted to `True`): Sets `/p:UseEnv=true` flag.

• **vcvars_ver** (Optional, Defaulted to `None`): Specifies the Visual Studio compiler toolset to use.

• **winsdk_version** (Optional, Defaulted to `None`): Specifies the version of the Windows SDK to use.

• **properties** (Optional, Defaulted to `None`): Dictionary with new properties, for each element in the dictionary `{name: value}` it will append a `/p:name="value"` option.

• **output_binary_log** (Optional, Defaulted to `None`): Sets `/bl` flag. If set to `True` then MSBuild will output a binary log file called `msbuild.binlog` in the working directory. It can also be used to set the name of log file like this `output_binary_log="my_log.binlog"`. This parameter is only supported starting from MSBuild version 15.3 and onwards.

• **property_file_name** (Optional, Defaulted to `None`): Sets `p:ForceImportBeforeCppTargets`. When `None` it will generate a file named `conan_build.props`. You can specify a different name for the generated properties file.

• **verbosity** (Optional, Defaulted to `None`): Sets the `/verbosity` flag to the specified verbosity level. Possible values are "quiet", "minimal", "normal", "detailed" and "diagnostic".

• **definitions** (Optional, Defaulted to `None`): Dictionary with additional compiler definitions to be applied during the build. Use a dictionary with the desired key and its value set to `None` to set a compiler definition with no value.
Note: The MSBuild() build helper will, before calling to **MSBuild**, call `tools.vcvars_command()` to adjust the environment according to the settings. When cross-building from x64 to x86 the toolchain by default is x86. If you want to use **amd64_x86** instead, set the environment variable `PreferredToolArchitecture=x64`.

### get_command()

Returns a string command calling **MSBuild**.

```python
def get_command(self, project_file, props_file_path=None, targets=None, upgrade_‐
→project=True,
   →build_type=None, arch=None, parallel=True, toolset=None,]
   →platforms=None,
   →use_env=False, properties=None, output_binary_log=None,
   →verbosity=None)
```

**Parameters:**

- **props_file_path** *(Optional, Defaulted to None)*: Path to a property file to be included in the compilation command. This parameter is automatically set by the `build()` method to set the runtime from settings.
- **Same parameters as the `build()` method.**

### get_version()

Static method that returns the version of MSBuild for the specified settings.

```python
def get_version(settings)
```

Result is returned in a `conans.model.Version` object as it is evaluated by the command line. It will raise an exception if it cannot resolve it to a valid result.

**Parameters:**

- **settings** *(Required)*: Conanfile settings. Use `self.settings`.

### 14.6.4 VisualStudioBuildEnvironment

Prepares the needed environment variables to invoke the Visual Studio compiler. Use it together with `tools.vcvars_command()`.

```python
from conans import ConanFile, VisualStudioBuildEnvironment

class ExampleConan(ConanFile):
    ...

def build(self):
    if self.settings.compiler == "Visual Studio":
        env_build = VisualStudioBuildEnvironment(self)
        with tools.environment_append(env_build.vars):
            vcvars = tools.vcvars_command(self.settings)
            self.run("%s && cl /EHsc hello.cpp" % vcvars)
            self.run("%s && lib hello.obj -OUT:hello.lib" % vcvars)
```

14.6. Build helpers
You can adjust the automatically filled attributes:

```python
def build(self):
    if self.settings.compiler == "Visual Studio":
        env_build = VisualStudioBuildEnvironment(self)
        env_build.include_paths.append("mycustom/directory/to/headers")
        env_build.lib_paths.append("mycustom/directory/to/libs")
        env_build.link_flags = []
        with tools.environment_append(env_build.vars):
            vcvars = tools.vcvars_command(self.settings)
            self.run("%s && cl /EHsc hello.cpp" % vcvars)
            self.run("%s && lib hello.obj -OUT:hello.lib" % vcvars)
```

Constructor

```python
class VisualStudioBuildEnvironment(object):
    def __init__(self, conanfile, with_build_type_flags=True)
```

Parameters:
- `with_build_type_flags` (Optional, Defaulted to `True`): If `True`, it adjusts the compiler flags according to the `build_type` setting. E.g: `-Zi`, `-Ob0`, `-Od`...

Environment variables

<table>
<thead>
<tr>
<th>NAME</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIB</td>
<td>Library paths separated with &quot;;&quot;</td>
</tr>
<tr>
<td>CL</td>
<td><code>/I</code> flags with include directories, Runtime <code>/MT, /MD...</code>, Definitions <code>/DXXX</code>, and any other C and CXX flags.</td>
</tr>
</tbody>
</table>

Attributes

**include_paths**

List with directories of include paths.

**lib_paths**

List with directories of libraries.

**defines**

List with definitions from requirements’ `cpp_info.defines`. 
runtime

List with directories from `settings.compiler.runtime).

flags

List with flags from requirements' `cpp_info.cflags`.

cxx_flags

List with cxx flags from requirements' `cpp_info.cxxflags`.

link_flags

List with linker flags from requirements' `cpp_info.sharedlinkflags` and `cpp_info.exelinkflags`.

std

This property contains the flag corresponding to the C++ standard. If you are still using the deprecated setting `cppstd` (see *How to manage C++ standard [EXPERIMENTAL]*)) and you are not providing any value for this setting, the property will be `None`

parallel

Defaulted to `False`.

Sets the flag `/MP` in order to compile the sources in parallel using cores found by `tools.cpu_count()`.

See also:
Read more about `tools.environment_append()`.

14.6.5 Meson

If you are using Meson Build as your build system, you can use the Meson build helper. Specially useful with the `pkg_config` that will generate the `.pc` files of our requirements, then Meson() build helper will locate them automatically.

```python
from conans import ConanFile, tools, Meson
import os

class ConanFileToolsTest(ConanFile):
    generators = "pkg_config"
    requires = "LIB_A/0.1@conan/stable"
    settings = "os", "compiler", "build_type"

    def build(self):
        meson = Meson(self)
        meson.configure(build_folder="build")
        meson.build()
```

14.6. Build helpers
Constructor

```python
class Meson(object):
    def __init__(self, conanfile, backend=None, build_type=None)
```

Parameters:
- **conanfile** (Required): Use `self` inside a `conanfile.py`.
- **backend** (Optional, Defaulted to None): Specify a backend to be used, otherwise it will use "Ninja".
- **build_type** (Optional, Defaulted to None): Force to use a build type, ignoring the value from the settings.

Methods

**configure()**

```python
def configure(self, args=None, defs=None, source_folder=None, build_folder=None, pkg_config_paths=None, cache_build_folder=None)
```

Configures Meson project with the given parameters.

Parameters:
- **args** (Optional, Defaulted to None): A list of additional arguments to be passed to the `configure` script. Each argument will be escaped according to the current shell. No extra arguments will be added if `args=None`.
- **defs** (Optional, Defaulted to None): A list of definitions.
- **source_folder** (Optional, Defaulted to None): Meson’s source directory where `meson.build` is located. The default value is the `self.source_folder`. Relative paths are allowed and will be relative to `self.source_folder`.
- **build_folder** (Optional, Defaulted to None): Meson’s output directory. The default value is the `self.build_folder` if None is specified. The Meson object will store `build_folder` internally for subsequent calls to `build()`.
- **pkg_config_paths** (Optional, Defaulted to None): A list containing paths to locate the pkg-config files (*.pc). If `None`, it will be set to `conanfile.build_folder`.
- **cache_build_folder** (Optional, Defaulted to None): Subfolder to be used as build folder when building the package in the local cache. This argument doesn’t have effect when the package is being built in user folder with `conan build` but overrides `build_folder` when working in the local cache. See `self.in_local_cache`.

**build()**

```python
def build(self, args=None, build_dir=None, targets=None)
```

Builds Meson project with the given parameters.

Parameters:
- **args** (Optional, Defaulted to None): A list of additional arguments to be passed to the `make` command. Each argument will be escaped according to the current shell. No extra arguments will be added if `args=None`.
**build_dir** (Optional, Defaulted to None): Build folder. If None, it will be set to conanfile.build_folder.

**targets** (Optional, Defaulted to None): A list of targets to be built. No targets will be added if targets=None.

### Example

A typical usage of the Meson build helper, if you want to be able to both execute `conan create` and also build your package for a library locally (in your user folder, not in the local cache), could be:

```python
from conans import ConanFile, Meson

class HelloConan(ConanFile):
    name = "Hello"
    version = "0.1"
    settings = "os", "compiler", "build_type", "arch"
    generators = "pkg_config"
    exports_sources = "src/*"
    requires = "zlib/1.2.11@conan/stable"

    def build(self):
        meson = Meson(self)
        meson.configure(source_folder="%s/src" % self.source_folder,
                        build_folder="build")
        meson.build()

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

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

Note the `pkg_config` generator, which generates `.pc` files (`zlib.pc` from the example above), which are understood by Meson to process dependencies information (no need for a `meson` generator).

The layout is:

```
<folder>
  | - conanfile.py
  | - src
  |   | - meson.build
  |   | - hello.cpp
  |   | - hello.h
```

And the `meson.build` could be as simple as:

```meson
project('hello',
       'cpp',
       version : '0.1.0'
       default_options : ['cpp_std=c++11'])
```

(continues on next page)
This allows, to create the package with `conan create` as well as to build the package locally:

```bash
$ cd <folder>
$ conan create . user/testing  # Now local build
$ mkdir build && cd build
$ conan install ..
$ conan build ..
```

### 14.6.6 RunEnvironment

The `RunEnvironment` helper prepares `PATH`, `LD_LIBRARY_PATH` and `DYLD_LIBRARY_PATH` environment variables to locate shared libraries and executables of your requirements at runtime.

**Warning:** The `RunEnvironment` is no longer needed, at least explicitly in `conanfile.py`. It has been integrated into the `self.run(..., run_environment=True)` argument. Check `self.run()`.

This helper is specially useful if:

- You are requiring packages with shared libraries and you are running some executable that needs those libraries.
- You have a requirement with some tool (executable) and you need it to be in the path.

```python
from conans import ConanFile, RunEnvironment

class ExampleConan(ConanFile):
    
    def build(self):
        env_build = RunEnvironment(self)
        with tools.environment_append(env_build.vars):
            self.run("....")
```

It sets the following environment variables:

<table>
<thead>
<tr>
<th>NAME</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>PATH</td>
<td>Containing all the requirements bin folders.</td>
</tr>
<tr>
<td>LD_LIBRARY_PATH</td>
<td>Containing all the requirements lib folders.</td>
</tr>
<tr>
<td>DYLD_LIBRARY_PATH</td>
<td>Containing all the requirements lib folders.</td>
</tr>
</tbody>
</table>

**Important:** Security restrictions might apply in OSX (read this thread), so the `DYLD_LIBRARY_PATH` environment variable is not directly transferred to the child process. In that case, you have to use it explicitly in your `conanfile.py`: 
def build(self):
    env_build = RunEnvironment(self)
    with tools.environment_append(env_build.vars):
        self.run('./myexetool')  # won't work, even if 'DYLD_LIBRARY_PATH' is in the env
        self.run('DYLD_LIBRARY_PATH=%s ./myexetool' % os.environ['DYLD_LIBRARY_PATH'])

This is already handled automatically by the `self.run(..., run_environment=True)` argument.

See also:
- Manage Shared Libraries with Environment Variables
- `tools.environment_append()`

### 14.7 Tools

Under the tools module there are several functions and utilities that can be used in Conan package recipes:

```python
from conans import ConanFile
from conans import tools

class ExampleConan(ConanFile):
    ...
```

#### 14.7.1 tools.cpu_count()

```python
def tools.cpu_count()
```

Returns the number of CPUs available, for parallel builds. If processor detection is not enabled, it will safely return 1. Can be overwritten with the environment variable `CONAN_CPU_COUNT` and configured in the `conan.conf`.

#### 14.7.2 tools.vcvars_command()

```python
def vcvars_command(settings, arch=None, compiler_version=None, force=False, vcvars_ver=None, winsdk_version=None)
```

Returns, for given settings, the command that should be called to load the Visual Studio environment variables for a certain Visual Studio version. It wraps the functionality of `vcvarsall` but does not execute the command, as that typically have to be done in the same command as the compilation, so the variables are loaded for the same subprocess. It will be typically used in the `build()` method, like this:

```python
from conans import tools

def build(self):
    if self.settings.build_os == 'Windows':
        vcvars = tools.vcvars_command(self.settings)
        build_command = ...
        self.run('"%s && configure %s" % (vcvars, " ".join(args)))
        self.run('"%s && %s %s" % (vcvars, build_command, " ".join(build_args)))
```
The `vcvars_command` string will contain something like `call "%vsXX0comntools%\..\..\VC/ vcvarsall.bat"` for the corresponding Visual Studio version for the current settings.

This is typically not needed if using CMake, as the `cmake` generator will handle the correct Visual Studio version.

If `arch` or `compiler_version` is specified, it will ignore the settings and return the command to set the Visual Studio environment for these parameters.

**Parameters:**

- **arch** (Optional, Defaulted to `None`): Will use `settings.arch`.
- **compiler_version** (Optional, Defaulted to `None`): Will use `settings.compiler.version`.
- **force** (Optional, Defaulted to `False`): Will ignore if the environment is already set for a different Visual Studio version.
- **winsdk_version** (Optional, Defaulted to `None`): Specifies the version of the Windows SDK to use.
- **vcvars_ver** (Optional, Defaulted to `None`): Specifies the Visual Studio compiler toolset to use.

**Note:** When cross-building from x64 to x86 the toolchain by default is x86. If you want to use `amd64_x86` instead, set the environment variable `PreferredToolArchitecture=x64`.

### 14.7.3 `tools.vcvars_dict()`

```python
vcvars_dict(settings, arch=None, compiler_version=None, force=False, filter_known_paths=False, vcsvars_ver=None, winsdk_version=None, only_diff=True)
```

Returns a dictionary with the variables set by the `tools.vcvars_command()` that can be directly applied to `tools.environment_append()`.

The values of the variables `INCLUDE`, `LIB`, `LIBPATH` and `PATH` will be returned as a list. When used with `tools.environment_append()`, the previous environment values that these variables may have will be appended automatically.

```python
from conans import tools

def build(self):
    env_vars = tools.vcvars_dict(self.settings)
    with tools.environment_append(env_vars):
        # Do something
```

**Parameters:**

- Same as `tools.vcvars_command()`.
- **filter_known_paths** (Optional, Defaulted to `False`): When True, the function will only keep the PATH entries that follow some known patterns, filtering all the non-Visual Studio ones. When False, it will keep the PATH will all the system entries.
- **only_diff** (Optional, Defaulted to `True`): When True, the command will return only the variables set by `vcvarsall` and not the whole environment. If `vcvars` modifies an environment variable by appending values to the old value (separated by `;`), only the new values will be returned, as a list.
14.7.4 tools.vcvars()

```python
vcvars(settings, arch=None, compiler_version=None, force=False, filter_known_paths=False)
```

**Note:** This context manager tool has no effect if used in a platform different from Windows.

This is a context manager that allows to append to the environment all the variables set by the `tools.vcvars_dict()`. You can replace `tools.vcvars_command()` and use this context manager to get a cleaner way to activate the Visual Studio environment:

```python
from conans import tools

def build(self):
    with tools.vcvars(self.settings):
        do_something()
```

14.7.5 tools.build_sln_command() [DEPRECATED]

**Warning:** This tool is deprecated and will be removed in Conan 2.0. Use `MSBuild()` build helper instead.

```python
def build_sln_command(settings, sln_path, targets=None, upgrade_project=True, build_type=None, arch=None, parallel=True, toolset=None, platforms=None, verbosity=None, definitions=None)
```

Returns the command to call `devenv` and `msbuild` to build a Visual Studio project. It’s recommended to use it with `tools.vcvars_command()`, so that the Visual Studio tools will be in path.

```python
from conans import tools

def build(self):
    build_command = build_sln_command(self.settings, "myfile.sln", targets=["SDL2_image"],
                                       upgrade_project=True, build_type=None, arch=None,
                                       parallel=True, toolset=None, platforms=None,
                                       verbosity=None, definitions=None)
    command = "$s && $s % (tools.vcvars_command(self.settings), build_command)
    self.run(command)
```

**Parameters:**

- **settings** (Required): Conanfile settings. Use “self.settings”.
- **sln_path** (Required): Visual Studio project file path.
- **targets** (Optional, Defaulted to None): List of targets to build.
- **upgrade_project** (Optional, Defaulted to True): If True, the project file will be upgraded if the project’s VS version is older than current. When `CONAN_SKIP_VS_PROJECTS_UPGRADE` environment variable is set to True/1, this parameter will be ignored and the project won’t be upgraded.
- **build_type** (Optional, Defaulted to None): Override the build type defined in the settings (`settings.build_type`).

14.7. Tools
• arch (Optional, Defaulted to None): Override the architecture defined in the settings (settings.arch).

• parallel (Optional, Defaulted to True): Enables Visual Studio parallel build with /m:X argument, where X is defined by CONAN_CPU_COUNT environment variable or by the number of cores in the processor by default.

• toolset (Optional, Defaulted to None): Specify a toolset. Will append a /p:PlatformToolset option.

• platforms (Optional, Defaulted to None): Dictionary with the mapping of archs/platforms from Conan naming to another one. It is useful for Visual Studio solutions that have a different naming in architectures. Example: platforms={"x86":"Win32"} (Visual solution uses “Win32” instead of “x86”). This dictionary will update the following default one:

```python
class msvc_arch:
    x86 = 'x86'
    x86_64 = 'x64'
    armv7 = 'ARM'
    armv8 = 'ARM64'
```

• verbosity (Optional, Defaulted to None): Specifies verbosity level (/verbosity: parameter).

• definitions (Optional, Defaulted to None): Dictionary with additional compiler definitions to be applied during the build. Use value of None to set compiler definition with no value.

### 14.7.6 tools.msvc_build_command() [DEPRECATED]

**Warning:** This tool is deprecated and will be removed in Conan 2.0. Use MSBuild().get_command() instead.

```python
def msvc_build_command(settings, sln_path, targets=None, upgrade_project=True, build_type=None, arch=None, parallel=True, force_vcvars=False, toolset=None, platforms=None):
    return tools.msvc_build_command(settings, sln_path, targets=targets, upgrade_project=upgrade_project, build_type=build_type, arch=arch, parallel=parallel, force_vcvars=force_vcvars, toolset=toolset, platforms=platforms)
```

Returns a string with a joint command consisting in setting the environment variables via vcvars.bat with the above tools.vcvars_command() function, and building a Visual Studio project with the tools.build_sln_command() [DEPRECATED] function.

**Parameters:**

- Same parameters as the above tools.build_sln_command() [DEPRECATED].

### 14.7.7 tools.unzip()

```python
def unzip(filename, destination=".", keep_permissions=False, pattern=None)
```

Function mainly used in source(), but could be used in build() in special cases, as when retrieving pre-built binaries from the Internet.

This function accepts .tar.gz, .tar, .tbz2, .tar.gz2, .tgz, .txz, tar.xz, and .zip files, and decompresses them into the given destination folder (the current one by default).
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.

```python
from conans import tools

tools.unzip("myfile.zip")
# or to extract in "myfolder" sub-folder
tools.unzip("myfile.zip", "myfolder")
```

You can keep the permissions of the files using the keep_permissions=True parameter.

```python
from conans import tools

tools.unzip("myfile.zip", "myfolder", keep_permissions=True)
```

Use pattern=None if you want to filter specific files and paths to decompress from the archive.

```python
from conans import tools

# Extract only files inside relative folder "small"
tools.unzip("bigfile.zip", pattern="small/*")
# Extract only txt files
tools.unzip("bigfile.zip", pattern="*.txt")
```

Parameters:

- **filename** (Required): File to be unzipped.
- **destination** (Optional, Defaulted to "."): Destination folder for unzipped files.
- **keep_permissions** (Optional, Defaulted to False): Keep permissions of files. WARNING: Can be dangerous if the zip was not created in a NIX system, the bits could produce undefined permission schema. Use only this option if you are sure that the zip was created correctly.
- **pattern** (Optional, Defaulted to None): Extract from the archive only paths matching the pattern. This should be a Unix shell-style wildcard. See fnmatch documentation for more details.

## 14.7.8 tools.untargz()

```python
def untargz(filename, destination=".", pattern=None)
```

Extract .tar.gz files (or in the family). This is the function called by the previous unzip() for the matching extensions, so generally not needed to be called directly, call unzip() instead unless the file had a different extension.

```python
from conans import tools

tools.untargz("myfile.tar.gz")
# or to extract in "myfolder" sub-folder
tools.untargz("myfile.tar.gz", "myfolder")
# or to extract only txt files
tools.untargz("myfile.tar.gz", pattern="*.txt")
```

Parameters:

- **filename** (Required): File to be unzipped.
- **destination** (Optional, Defaulted to "."): Destination folder for untargzed files.
• **pattern** (Optional, Defaulted to `None`): Extract from the archive only paths matching the pattern. This should be a Unix shell-style wildcard. See `fnmatch` documentation for more details.

### 14.7.9 tools.get()

```python
def get(url, filename='', md5='', sha1='', sha256='', keep_permissions=False, 
       pattern=None, verify=True, retry=2, retry_wait=5, overwrite=False, auth=None, headers=None)
```

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`.

```python
from conans import tools
tools.get("http://url/file", md5='d2da0cd0756cd9da6560b9a56016a0cb')
# also, specify a destination folder
tools.get("http://url/file", destination="subfolder")
```

**Parameters:**

- **url** (Required): URL to download.
- **filename** (Optional, Defaulted to `""`): Specify the name of the compressed file if it cannot be deduced from the URL.
- **md5** (Optional, Defaulted to `""`): MD5 hash code to check the downloaded file.
- **sha1** (Optional, Defaulted to `""`): SHA1 hash code to check the downloaded file.
- **sha256** (Optional, Defaulted to `""`): SHA256 hash code to check the downloaded file.
- **keep_permissions** (Optional, Defaulted to `False`): Propagates the parameter to `tools.unzip()`.
- **pattern** (Optional, Defaulted to `None`): Propagates the parameter to `tools.unzip()`.
- **verify** (Optional, Defaulted to `True`): When False, disables https certificate validation.
- **retry** (Optional, Defaulted to `2`): Number of retries in case of failure. Default is overridden by `general.retry` in the `conan.conf` file or an env variable `CONAN_RETRY`.
- **retry_wait** (Optional, Defaulted to `5`): Seconds to wait between download attempts. Default is overridden by `general.retry_wait` in the `conan.conf` file or an env variable `CONAN_RETRY_WAIT`.
- **overwrite** (Optional, Defaulted to `False`): When True Conan will overwrite the destination file if it exists. Otherwise it will raise.
- **auth** (Optional, Defaulted to `None`): A tuple of user, password can be passed to use HTTPBasic authentication. This is passed directly to the `requests` Python library. Check here other uses of the `auth` parameter: https://2.python-requests.org/en/master/user/authentication/
- **headers** (Optional, Defaulted to `None`): A dictionary with additional headers.

### 14.7.10 tools.get_env()

```python
def get_env(env_key, default=None, environment=None)
```
Parses an environment and cast its value against the default type passed as an argument. Following Python conventions, returns default if env_key is not defined.

This is a usage example with an environment variable defined while executing Conan:

```bash
$ TEST_ENV="1" conan <command> ...
```

```python
from conans import tools

tools.get_env("TEST_ENV")  # returns "1", returns current value
tools.get_env("TEST_ENV_NOT_DEFINED")  # returns None, TEST_ENV_NOT_DEFINED not declared
tools.get_env("TEST_ENV_NOT_DEFINED", [])  # returns [], TEST_ENV_NOT_DEFINED not declared
tools.get_env("TEST_ENV", "2")  # returns "2"
tools.get_env("TEST_ENV", False)  # returns True (default value is boolean)
tools.get_env("TEST_ENV", 2)  # returns 2
tools.get_env("TEST_ENV", 2.0)  # returns 2.0
tools.get_env("TEST_ENV", [])  # returns ["1"]
```

**Parameters:**

- **env_key** (Required): environment variable name.
- **default** (Optional, Defaulted to None): default value to return if not defined or cast value against.
- **environment** (Optional, Defaulted to None): os.environ if None or environment dictionary to look for.

### 14.7.11 tools.download()

```python
def download(url, filename, verify=True, out=None, retry=2, retry_wait=5, overwrite=False, auth=None, headers=None)
```

Retrieves a file from a given URL into a file with a given filename. It uses certificates from a list of known verifiers for https downloads, but this can be optionally disabled.

```python
from conans import tools

tools.download("http://someurl/somefile.zip", "myfilename.zip")
# to disable verification:
tools.download("http://someurl/somefile.zip", "myfilename.zip", verify=False)
# to retry the download 2 times waiting 5 seconds between them
tools.download("http://someurl/somefile.zip", "myfilename.zip", retry=2, retry_wait=5)
# Use https basic authentication
tools.download("http://someurl/somefile.zip", "myfilename.zip", auth=('user', 'password'))
# Pass some header
tools.download("http://someurl/somefile.zip", "myfilename.zip", headers={'Myheader': 'My value'})
```

**Parameters:**
• **url** (Required): URL to download

• **filename** (Required): Name of the file to be created in the local storage

• **verify** (Optional, Defaulted to True): When False, disables HTTPS certificate validation.

• **out** (Optional, Defaulted to None): An object with a write() method can be passed to get the output. stdout will use if not specified.

• **retry** (Optional, Defaulted to 2): Number of retries in case of failure. Default is overridden by `general.retry` in the `conan.conf` file or an env variable `CONAN_RETRY`.

• **retry_wait** (Optional, Defaulted to 5): Seconds to wait between download attempts. Default is overridden by `general.retry_wait` in the `conan.conf` file or an env variable `CONAN_RETRY_WAIT`.

• **overwrite** (Optional, Defaulted to False): When True, Conan will overwrite the destination file if exists. Otherwise it will raise an exception.

• **auth** (Optional, Defaulted to None): A tuple of user and password to use HTTPBasic authentication. This is used directly in the `requests` Python library. Check other uses here: https://2.python-requests.org/en/master/user/authentication/

• **headers** (Optional, Defaulted to None): A dictionary with additional headers.

### 14.7.12 tools.ftp_download()

```python
def ftp_download(ip, filename, login='', password='')
```

Retrieves a file from an FTP server. This doesn’t support SSL, but you might implement it yourself using the standard Python FTP library.

```python
from conans import tools

def source(self):
    tools.ftp_download('ftp.debian.org', "debian/README")
    self.output.info(load("README"))
```

**Parameters:**

• **ip** (Required): The IP or address of the ftp server.

• **filename** (Required): The filename, including the path/folder where it is located.

• **login** (Optional, Defaulted to ''): Login credentials for the ftp server.

• **password** (Optional, Defaulted to ''): Password credentials for the ftp server.

### 14.7.13 tools.replace_in_file()

```python
def replace_in_file(file_path, search, replace, strict=True)
```

This function is useful for a simple “patch” or modification of source files. A typical use would be to augment some library existing `CMakeLists.txt` in the `source()` method of a `conanfile.py`, so it uses Conan dependencies without forking or modifying the original project:

```python
from conans import tools

def source(self):
    # (continues on next page)
```
# get the sources from somewhere

tools.replace_in_file("hello/CMakeLists.txt", "PROJECT(MyHello)",
                 '''PROJECT(MyHello)
include(${CMAKE_BINARY_DIR}/conanbuildinfo.cmake)
conan_basic_setup()'''
)

Parameters:

- **file_path** (Required): File path of the file to perform the replace in.
- **search** (Required): String you want to be replaced.
- **replace** (Required): String to replace the searched string.
- **strict** (Optional, Defaulted to `True`): If `True`, it raises an error if the searched string is not found, so nothing is actually replaced.

### 14.7.14 `tools.replace_path_in_file()`

```python
def replace_path_in_file(file_path, search, replace, strict=True, windows_paths=None)
```

Replace a path in a file with another string. In Windows, it will match the path even if the casing and the path separator doesn’t match.

```python
from conans import tools
def build(self):
    tools.replace_path_in_file("hello/somefile.cmake", "c:\Some/PATH/to\File.txt",
                            "PATTERN/file.txt")
```

Parameters:

- **file_path** (Required): File path of the file to perform the replace in.
- **search** (Required): String with the path you want to be replaced.
- **replace** (Required): String to replace the searched path.
- **strict** (Optional, Defaulted to `True`): If `True`, it raises an error if the search string is not found and nothing is actually replaced.
- **windows_paths** (Optional, Defaulted to `None`): Controls whether the casing of the path and the different directory separators are taken into account:
  - `None`: Only when Windows operating system is detected.
  - `False`: Deactivated, it will match exact patterns (like `tools.replace_in_file()`).
  - `True`: Always activated, irrespective of the detected operating system.

### 14.7.15 `tools.run_environment()`

```python
def run_environment(conanfile)
```

Context manager that sets temporary environment variables set by `RunEnvironment`.

---

14.7. Tools 413
14.7.16 tools.check_with_algorithm_sum()

```python
def check_with_algorithm_sum(algorithm_name, file_path, signature)
```

Useful to check that some downloaded file or resource has a predefined hash, so integrity and security are guaranteed. Something that could be typically done in `source()` method after retrieving some file from the internet.

**Parameters:**

- **algorithm_name** (Required): Name of the algorithm to be checked.
- **file_path** (Required): File path of the file to be checked.
- **signature** (Required): Hash code that the file should have.

There are specific functions for common algorithms:

```python
def check_sha1(file_path, signature)
def check_md5(file_path, signature)
def check_sha256(file_path, signature)
```

For example:

```python
from conans import tools
tools.check_sha1("myfile.zip", "eb599ec83d383f0f25691c184f656d40384f9435")
```

Other algorithms are also possible, as long as are recognized by python `hashlib` implementation, via `hashlib.new(algorithm_name)`. The previous is equivalent to:

```python
from conans import tools
tools.check_with_algorithm_sum("sha1", "myfile.zip",
                            "eb599ec83d383f0f25691c184f656d40384f9435")
```

14.7.17 tools.patch()

```python
def patch(base_path=None, patch_file=None, patch_string=None, strip=0, output=None)
```

Applies a patch from a file or from a string into the given path. The patch should be in diff (unified diff) format. To be used mainly in the `source()` method.

```python
from conans import tools
tools.patch(patch_file="file.patch")
# from a string:
patch_content = " real patch content ..."
tools.patch(patch_string=patch_content)
# to apply in subfolder
tools.patch(base_path=mysubfolder, patch_string=patch_content)
```

If the patch to be applied uses alternate paths that have to be stripped like this example:

```bash
--- old_path/text.txt	2016-01-25 17:57:11.452848309 +0100
@@ -1 +1 @@
(continues on next page)```
Then, the number of folders to be stripped from the path can be specified:

```python
from conans import tools
tools.patch(patch_file="file.patch", strip=1)
```

**Parameters:**

- `base_path` (Optional, Defaulted to `None`): Base path where the patch should be applied.
- `patch_file` (Optional, Defaulted to `None`): Patch file that should be applied.
- `patch_string` (Optional, Defaulted to `None`): Patch string that should be applied.
- `strip` (Optional, Defaulted to `0`): Number of folders to be stripped from the path.
- `output` (Optional, Defaulted to `None`): Stream object.

### 14.7.18 `tools.environment_append()`

```python
def environment_append(env_vars)
```

This is a context manager that allows to temporary use environment variables for a specific piece of code in your conanfile:

```python
from conans import tools
def build(self):
    with tools.environment_append({"MY_VAR": "3", "CXX": "/path/to/cxx", "CPPFLAGS": "", "CPPFLAGS": None}):
        do_something()
```

The environment variables will be overridden if the value is a string, while it will be prepended if the value is a list. Additionally, if value is `None`, the given environment variable is unset (In the previous example, `CPPFLAGS` environment variable will be unset), and in case variable wasn’t set prior to the invocation, it has no effect on the given variable (`CPPFLAGS`). When the context manager block ends, the environment variables will recover their previous state.

**Parameters:**

- `env_vars` (Required): Dictionary object with environment variable name and its value.

### 14.7.19 `tools.chdir()`

```python
def chdir(newdir)
```

This is a context manager that allows to temporary change the current directory in your conanfile:

```python
from conans import tools
def build(self):
    with tools.chdir("./subdir"):
        do_something()
```
Parameters:

- `newdir` (Required): Directory path name to change the current directory.

### 14.7.20 tools.pythonpath()

*Warning*: This way of reusing python code from other recipes can be improved via *Python requires: reusing code [EXPERIMENTAL]*.

This tool is automatically applied in the conanfile methods unless `apply_env` is deactivated, so any PYTHONPATH inherited from the requirements will be automatically available.

```python
def pythonpath(conanfile)
```

This is a context manager that allows to load the PYTHONPATH for dependent packages, create packages with Python code and reuse that code into your own recipes.

For example:

```python
from conans import tools
def build(self):
    with tools.pythonpath(self):
        from module_name import whatever
        whatever.do_something()
```

When the `apply_env` is activated (default) the above code could be simplified as:

```python
from conans import tools
def build(self):
    from module_name import whatever
    whatever.do_something()
```

For that to work, one of the dependencies of the current recipe, must have a `module_name` file or folder with a `whatever` file or object inside, and should have declared in its `package_info()`:

```python
from conans import tools
def package_info(self):
    self.env_info.PYTHONPATH.append(self.package_folder)
```

Parameters:

- `conanfile` (Required): Current ConanFile object.

### 14.7.21 tools.no_op()

```python
def no_op()
```

Context manager that performs nothing. Useful to condition any other context manager to get a cleaner code:
from conans import tools

def build(self):
    with tools.chdir("some_dir") if self.options.myoption else tools.no_op():
        # if not self.options.myoption, we are not in the "some_dir"
        pass

14.7.22 tools.human_size()

```python
def human_size(size_bytes)
```

Will return a string from a given number of bytes, rounding it to the most appropriate unit: GB, MB, KB, etc. It is mostly used by the Conan downloads and unzip progress.

```python
from conans import tools

tools.human_size(1024)
>> 1.0KB
```

Parameters:
- **size_bytes** (Required): Number of bytes.

14.7.23 tools.OSInfo and tools.SystemPackageTool

These are helpers to install system packages. Check `system_requirements()`.

14.7.24 tools.cross_building()

```python
def cross_building(settings, self_os=None, self_arch=None)
```

Reading the settings and the current host machine it returns `True` if we are cross building a Conan package:

```python
from conans import tools

if tools.cross_building(self.settings):
    # Some special action
```

Parameters:
- **self_os** (Optional, Defaulted to `None`): Current operating system where the build is being done.
- **self_arch** (Optional, Defaulted to `None`): Current architecture where the build is being done.

14.7.25 tools.get_gnu_triplet()

```python
def get_gnu_triplet(os_, arch, compiler=None)
```

Returns string with GNU like `<machine>-<vendor>-<op_system>` triplet.

Parameters:
• **os_** (Required): Operating system to be used to create the triplet.
• **arch** (Required): Architecture to be used to create the triplet.
• **compiler** (Optional, Defaulted to None): Compiler used to create the triplet (only needed for Windows).

### 14.7.26 tools.run_in_windows_bash()

```python
def run_in_windows_bash(conanfile, bashcmd, cwd=None, subsystem=None, msys_mingw=True, ← env=None, with_login=True)
```

Runs a UNIX command inside a bash shell. It requires to have “bash” in the path. Useful to build libraries using `configure` and `make` in Windows. Check Windows subsytems section.

You can customize the path of the bash executable using the environment variable `CONAN_BASH_PATH` or the `conan.conf` `bash_path` variable to change the default bash location.

```python
from conans import tools
command = "pwd"
tools.run_in_windows_bash(self, command)  # self is a conanfile instance
```

**Parameters:**

- **conanfile** (Required): Current ConanFile object.
- **bashcmd** (Required): String with the command to be run.
- **cwd** (Optional, Defaulted to None): Path to directory where to apply the command from.
- **subsystem** (Optional, Defaulted to None): Used to escape the command according to the specified subsystem.
- **msys_mingw** (Optional, Defaulted to True): If the specified subsystem is MSYS2, will start it in MinGW mode (native windows development).
- **env** (Optional, Defaulted to None): You can pass a dictionary with environment variable to be applied at first place so they will have more priority than others.
- **with_login** (Optional, Defaulted to True): Pass the `--login` flag to `bash` command. This might come handy when you don’t want to create a fresh user session for running the command.

### 14.7.27 tools.get_cased_path()

```python
def get_cased_path(abs_path)
```

This function converts a case-insensitive absolute path to a case-sensitive one. That is, with the real cased characters. Useful when using Windows subsystems where the file system is case-sensitive.

### 14.7.28 tools.detected_os()

```python
def detected_os()
```

It returns the recognized OS name e.g “Macos”, “Windows”. Otherwise it will return the value from `platform.system()`.
14.7.29 tools.remove_from_path()

```python
remove_from_path(command)
```

This is a context manager that allows you to remove a tool from the PATH. Conan will locate the executable (using `tools.which()`) and will remove from the PATH the directory entry that contains it. It’s not necessary to specify the extension.

```python
from conans import tools
with tools.remove_from_path("make"):  
    self.run("some command")
```

14.7.30 tools.unix_path()

```python
def unix_path(path, path_flavor=None)
```

Used to translate Windows paths to MSYS/CYGWIN Unix paths like `c/users/path/to/file`.

**Parameters:**
- `path` (Required): Path to be converted.
- `path_flavor` (Optional, Defaulted to None, will try to autodetect the subsystem): Type of Unix path to be returned. Options are MSYS, MSYS2, CYGWIN, WSL and SFU.

14.7.31 tools.escape_windows_cmd()

```python
def escape_windows_cmd(command)
```

Useful to escape commands to be executed in a windows bash (msys2, cygwin etc).

- Adds escapes so the argument can be unpacked by `CommandLineToArgvW()`.
- Adds escapes for `cmd.exe` so the argument survives to `cmd.exe`’s substitutions.

**Parameters:**
- `command` (Required): Command to execute.

14.7.32 tools.sha1sum(), sha256sum(), md5sum()

```python
def md5sum(file_path)
def sha1sum(file_path)
def sha256sum(file_path)
```

Return the respective hash or checksum for a file.

```python
from conans import tools
md5 = tools.md5sum("myfilepath.txt")
shal = tools.sha1sum("myfilepath.txt")
```

**Parameters:**
- `file_path` (Required): Path to the file.

14.7. Tools
14.7.33 tools.md5()

```python
def md5(content)
```

Returns the MD5 hash for a string or byte object.

```python
from conans import tools
md5 = tools.md5("some string, not a file path")
```

Parameters:
- **content** (Required): String or bytes to calculate its md5.

14.7.34 tools.save()

```python
def save(path, content, append=False)
```

Utility function to save files in one line. It will manage the open and close of the file and creating directories if necessary.

```python
from conans import tools
tools.save("otherfile.txt", "contents of the file")
```

Parameters:
- **path** (Required): Path to the file.
- **content** (Required): Content that should be saved into the file.
- **append** (Optional, Defaulted to False): If True, it will append the content.

14.7.35 tools.load()

```python
def load(path, binary=False)
```

Utility function to load files in one line. It will manage the open and close of the file, and load binary encodings. Returns the content of the file.

```python
from conans import tools
content = tools.load("myfile.txt")
```

Parameters:
- **path** (Required): Path to the file.
- **binary** (Optional, Defaulted to False): If True, it reads the the file as binary code.

14.7.36 tools.mkdir(), tools.rmdir()
Utility functions to create/delete a directory. The existence of the specified directory is checked, so `mkdir()` will do nothing if the directory already exists and `rmdir()` will do nothing if the directory does not exist.

This makes it safe to use these functions in the `package()` method of a `conanfile.py` when `no_copy_source=True`.

```python
from conans import tools

tools.mkdir("mydir")  # Creates mydir if it does not already exist
tools.mkdir("mydir")  # Does nothing

tools.rmdir("mydir")  # Deletes mydir
tools.rmdir("mydir")  # Does nothing
```

**Parameters:**
- `path` (Required): Path to the directory.

### 14.7.37 tools.which()

```python
def which(filename)
```

Returns the path to a specified executable searching in the `PATH` environment variable. If not found, it returns `None`. This tool also looks for filenames with following extensions if no extension provided:

- `.com`, `.exe`, `.bat .cmd` for Windows.
- `.sh` if not Windows.

```python
from conans import tools

abs_path_make = tools.which("make")
```

**Parameters:**
- `filename` (Required): Name of the executable file. It doesn’t require the extension of the executable.

### 14.7.38 tools.unix2dos()

```python
def unix2dos(filepath)
```

Converts line breaks in a text file from Unix format (LF) to DOS format (CRLF).

```python
from conans import tools

tools.unix2dos("project.dsp")
```

**Parameters:**
- `filepath` (Required): The file to convert.

### 14.7.39 tools.dos2unix()

```python
def dos2unix(filepath)
```

**Parameters:**
- `filepath` (Required): The file to convert.
Converts line breaks in a text file from DOS format (CRLF) to Unix format (LF).

```python
from conans import tools
tools.dos2unix("dosfile.txt")
```

Parameters:
- **filepath** (Required): The file to convert.

### 14.7.40 tools.touch()

```python
def touch(fname, times=None)
```

Updates the timestamp (last access and last modification times) of a file. This is similar to Unix’ `touch` command except that this one fails if the file does not exist.

Optionally, a tuple of two numbers can be specified, which denotes the new values for the last access and last modified times respectively.

```python
from conans import tools
import time
tools.touch("myfile")  # Sets atime and mtime to the current time
tools.touch("myfile", (time.time(), time.time()))  # Similar to above
tools.touch("myfile", (time.time(), 1))  # Modified long, long ago
```

Parameters:
- **fname** (Required): File name of the file to be touched.
- **times** (Optional, Defaulted to None): Tuple with ‘last access’ and ‘last modified’ times.

### 14.7.41 tools.relative_dirs()

```python
def relative_dirs(path)
```

Recursively walks a given directory (using `os.walk()`) and returns a list of all contained file paths relative to the given directory.

```python
from conans import tools
tools.relative_dirs("mydir")
```

Parameters:
- **path** (Required): Path of the directory.

### 14.7.42 tools.vswhere()

```python
def vswhere(all_=False, prerelease=False, products=None, requires=None, version="", latest=False, legacy=False, property_="", nologo=True)
```
Wrapper of vswhere tool to look for details of Visual Studio installations. Its output is always a list with a dictionary for each installation found.

```python
from conans import tools
vs_legacy_installations = tool.vswhere(legacy=True)
```

Parameters:

- **all_** (Optional, Defaulted to False): Finds all instances even if they are incomplete and may not launch.
- **prerelease** (Optional, Defaulted to False): Also searches prereleases. By default, only releases are searched.
- **products** (Optional, Defaulted to None): List of one or more product IDs to find. Defaults to Community, Professional, and Enterprise. Specify ["*"] by itself to search all product instances installed.
- **requires** (Optional, Defaulted to None): List of one or more workload or component IDs required when finding instances. See https://docs.microsoft.com/en-us/visualstudio/install/workload-and-component-ids?view=vs-2017 listing all workload and component IDs.
- **version** (Optional, Defaulted to None): A version range of instances to find. Example: "[15.0,16.0)" will find versions 15.*.
- **latest** (Optional, Defaulted to False): Return only the newest version and last installed.
- **legacy** (Optional, Defaulted to False): Also searches Visual Studio 2015 and older products. Information is limited. This option cannot be used with either products or requires parameters.
- **property_** (Optional, Defaulted to None): The name of a property to return. Use delimiters .. /, or _ to separate object and property names. Example: "properties.nickname" will return the “nickname” property under “properties”.
- **nologo** (Optional, Defaulted to True): Do not show logo information.

### 14.7.43 tools.vs_comntools()

```python
def vs_comntools(compiler_version)
```

Returns the value of the environment variable VS<compiler_version>.0COMNTOOLS for the compiler version indicated.

```python
from conans import tools
vs_path = tools.vs_comntools("14")
```

Parameters:

- **compiler_version** (Required): String with the version number: "14", "12"...

### 14.7.44 tools.vs_installation_path()

```python
def vs_installation_path(version, preference=None)
```

Returns the Visual Studio installation path for the given version. It uses tools.vswhere() and tools.vs_comntools(). It will also look for the installation paths following CONAN_VS_INSTALLATION_PREFERENCE environment variable or the preference parameter itself. If the tool is not able to return the path it will return None.
from conans import tools

def vs_path_2017 = tools.vs_installation_path("15", preference=["Community", "BuildTools", "Professional", "Enterprise"])

Parameters:
- **version** (Required): Visual Studio version to locate. Valid version numbers are strings: "10", "11", "12", "13", "14", "15"...
- **preference** (Optional, Defaulted to None): Set to value of CONAN_VS_INSTALLATION_PREFERENCE or defaulted to ["Enterprise", "Professional", "Community", "BuildTools"]. If only set to one type of preference, it will return the installation path only for that Visual type and version, otherwise None.

14.7.45 tools.replace_prefix_in_pc_file()

```python
def replace_prefix_in_pc_file(pc_file, new_prefix)
```

Replaces the prefix variable in a package config file .pc with the specified value.

```python
from conans import tools
lib_b_path = self.deps_cpp_info["libB"].rootpath
tools.replace_prefix_in_pc_file("libB.pc", lib_b_path)
```

Parameters:
- **pc_file** (Required): Path to the pc file
- **new_prefix** (Required): New prefix variable value (Usually a path pointing to a package).

See also:
Check section *pkg-config and .pc files* to know more.

14.7.46 tools.collect_libs()

```python
def 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.

```python
from conans import tools
def package_info(self):
    self.cpp_info.libdirs = ["lib", "other_libdir"]  # Default value is 'lib'
    self.cpp_info.libs = tools.collect_libs(self)
```

For UNIX libraries staring with **lib**, like *libmath.a*, this tool will collect the library name **math**.

Parameters:
- **conanfile** (Required): A ConanFile object to get the package_folder and cpp_info.
- **folder** (Optional, Defaulted to `None`): String indicating the subfolder name inside `conanfile.package_folder` where the library files are.

**Warning:** This tool collects the libraries searching directly inside the package folder and returns them in no specific order. If libraries are inter-dependent, then `package_info()` method should order them to achieve correct linking order.

### 14.7.47 tools.PkgConfig()

```python
class PkgConfig(library, pkg_config_executable="pkg-config", static=False, msvc_syntax=False, variables=None, print_errors=True)
```

Wrapper of the `pkg-config` tool.

```python
from conans import tools

with environment_append({'PKG_CONFIG_PATH': tmp_dir}):
    pkg_config = PkgConfig("libastral")
    print(pkg_config.cflags)
    print(pkg_config.cflags_only_I)
    print(pkg_config.variables)
```

**Parameters of the constructor:**

- **library** (Required): Library (package) name, such as `libastral`.
- **pkg_config_executable** (Optional, Defaulted to "pkg-config"): Specify custom pkg-config executable (e.g., for cross-compilation).
- **static** (Optional, Defaulted to `False`): Output libraries suitable for static linking (adds --static to `pkg-config` command line).
- **msvc_syntax** (Optional, Defaulted to `False`): MSVC compatibility (adds --msvc-syntax to `pkg-config` command line).
- **variables** (Optional, Defaulted to `None`): Dictionary of pkg-config variables (passed as --define-variable=VARIABLENAME=VARIABLEVALUE).
- **print_errors** (Optional, Defaulted to `True`): Output error messages (adds –print-errors)

**Properties:**

<table>
<thead>
<tr>
<th>PROPERTY</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>.cflags</td>
<td>get all pre-processor and compiler flags</td>
</tr>
<tr>
<td>.cflags_only_I</td>
<td>get -I flags</td>
</tr>
<tr>
<td>.cflags_only_other</td>
<td>get cflags not covered by the cflags-only-I option</td>
</tr>
<tr>
<td>.libs</td>
<td>get all linker flags</td>
</tr>
<tr>
<td>.libs_only_L</td>
<td>get -L flags</td>
</tr>
<tr>
<td>.libs_only_l</td>
<td>get -l flags</td>
</tr>
<tr>
<td>.libs_only_other</td>
<td>get other libs (e.g., -pthread)</td>
</tr>
<tr>
<td>.provides</td>
<td>get which packages the package provides</td>
</tr>
<tr>
<td>.requires</td>
<td>get which packages the package requires</td>
</tr>
<tr>
<td>.requires_private</td>
<td>get packages the package requires for static linking</td>
</tr>
<tr>
<td>.variables</td>
<td>get list of variables defined by the module</td>
</tr>
</tbody>
</table>
Warning: This is an experimental feature subject to breaking changes in future releases.

class Git(folder=None, verify_ssl=True, username=None, password=None, force_english=True, runner=None):

Wrapper of the git tool.

Parameters of the constructor:

- **folder** (Optional, Defaulted to `None`): Specify a subfolder where the code will be cloned. If not specified it will clone in the current directory.
- **verify_ssl** (Optional, Defaulted to `True`): Verify SSL certificate of the specified `url`.
- **username** (Optional, Defaulted to `None`): When present, it will be used as the login to authenticate with the remote.
- **password** (Optional, Defaulted to `None`): When present, it will be used as the password to authenticate with the remote.
- **force_english** (Optional, Defaulted to `True`): The encoding of the tool will be forced to use `en_US.UTF-8` to ease the output parsing.
- **runner** (Optional, Defaulted to `None`): By default `subprocess.check_output` will be used to invoke the git tool.

Methods:

- **run(command)**: Run any “git” command, e.g., `run("status")`
- **get_url_with_credentials(url)**: Returns the passed URL but containing the `username` and `password` in the URL to authenticate (only if `username` and `password` is specified)
- **clone(url, branch=None)**: Clone a repository. Optionally you can specify a branch. Note: If you want to clone a repository and the specified `folder` already exist you have to specify a `branch`.
- **checkout(element, submodule=None)**: Checkout a branch, commit or tag given by `element`. Argument `submodule` can get values in `shallow` or `recursive` to instruct what to do with submodules.
- **get_remote_url(remote_name=None)**: Returns the remote URL of the specified remote. If not `remote_name` is specified `origin` will be used.
- **get_qualified_remote_url()**: Returns the remote url (see `get_remote_url()`) but with forward slashes if it is a local folder.
- **get_revision(), get_commit()**: Gets the current commit hash.
- **get_branch()**: Gets the current branch.
- **get_tag()**: Gets the current checkout tag (`git describe --exact-match --tags`) and returns `None` if not in a tag.
- **excluded_files()**: Gets a list of the files and folders that would be excluded by `.gitignore` file.
- **is_local_repository()**: Returns `True` if the remote is a local folder.
- **is_pristine()**: Returns `True` if there aren’t modified or uncommitted files in the working copy.
- **get_repo_root()**: Returns the root folder of the working copy.
- **get_commit_message()**: Returns the latest log message
14.7.49 tools.SVN()

**Warning:** This is an experimental feature subject to breaking changes in future releases.

```python
class SVN(folder=None, verify_ssl=True, username=None, password=None,
        force_english=True, runner=None):
```

Wrapper of the `svn` tool.

**Parameters of the constructor:**

- `folder` (Optional, Defaulted to `None`): Specify a subfolder where the code will be cloned. If not specified it will clone in the current directory.
- `verify_ssl` (Optional, Defaulted to `True`): Verify SSL certificate of the specified url.
- `username` (Optional, Defaulted to `None`): When present, it will be used as the login to authenticate with the remote.
- `password` (Optional, Defaulted to `None`): When present, it will be used as the password to authenticate with the remote.
- `force_english` (Optional, Defaulted to `True`): The encoding of the tool will be forced to use en_US. UTF-8 to ease the output parsing.
- `runner` (Optional, Defaulted to `None`): By default `subprocess.check_output` will be used to invoke the `svn` tool.

**Methods:**

- `version()`: Retrieve version from the installed SVN client.
- `run(command)`: Run any “svn” command, e.g., `run("status")`
- `get_url_with_credentials(url)`: Return the passed url but containing the `username` and `password` in the URL to authenticate (only if `username` and `password` is specified)
- `checkout(url, revision="HEAD")`: Checkout the revision number given by `revision` from the specified `url`.
- `update(revision="HEAD")`: Update working copy to revision number given by `revision`.
- `get_remote_url()`: Returns the remote url of working copy.
- `get_qualified_remote_url()`: Returns the remote url of the working copy with the peg revision appended to it.
- `get_revision()`: Gets the current revision number from the repo server.
- `get_last_changed_revision(use_wc_root=True)`: Returns the revision number corresponding to the last changed item in the working folder (use_wc_root=False) or in the working copy root (use_wc_root=True).
- `get_branch()`: Tries to deduce the branch name from the standard SVN layout. Will raise if cannot resolve it.
- `get_tag()`: Tries to deduce the tag name from the standard SVN layout and returns the current tag name. Otherwise it will return `None`.
- `excluded_files()`: Gets a list of the files and folders that are marked to be ignored.
- `is_local_repository()`: Returns `True` if the remote is a local folder.
• **is_pristine()**: Returns `True` if there aren’t modified or uncommitted files in the working copy.

• **get_repo_root()**: Returns the root folder of the working copy.

• **get_revision_message()**: Returns the latest log message.

**Warning:** SVN allows to checkout a subdirectory of the remote repository, take into account that the return value of some of these functions may depend on the root of the working copy that has been checked out.

### 14.7.50 tools.is_apple_os()

```python
def is_apple_os(os_)
```

Returns `True` if OS is an Apple one: macOS, iOS, watchOS or tvOS.

**Parameters:**

- `os_` (Required): OS to perform the check. Usually this would be `self.settings.os`.

### 14.7.51 tools.to_apple_arch()

```python
def to_apple_arch(arch)
```

Converts Conan style architecture into Apple style architecture.

**Parameters:**

- `arch` (Required): arch to perform the conversion. Usually this would be `self.settings.arch`.

### 14.7.52 tools.apple_sdk_name()

```python
def apple_sdk_name(settings)
```

Returns proper SDK name suitable for OS and architecture you are building for (considering simulators).

**Parameters:**

- `settings` (Required): Conanfile settings.

### 14.7.53 tools.apple_deployment_target_env()

```python
def apple_deployment_target_env(os_, os_version)
```

Environment variable name which controls deployment target: `MACOSX_DEPLOYMENT_TARGET`, `IOS_DEPLOYMENT_TARGET`, `WATCHOS_DEPLOYMENT_TARGET` or `TVOS_DEPLOYMENT_TARGET`.

**Parameters:**

- `os_version` (Required): OS version.
### 14.7.54 tools.apple_deployment_target_flag()

```python
def apple_deployment_target_flag(os_, os_version)
```

Compiler flag name which controls deployment target. For example: `-mappletvos-version-min=9.0`

**Parameters:**
- `os_version` (Required): OS version.

### 14.7.55 tools.XCRun()

```python
class XCRun(object):
    def __init__(self, settings, sdk=None):
```

XCRun wrapper used to get information for building.

**Properties:**
- `sdk_path`: Obtain SDK path (a.k.a. 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).

### 14.7.56 tools.latest_vs_version_installed()

```python
def latest_vs_version_installed()
```

Returns a string with the major version of latest Microsoft Visual Studio available on machine. If no Microsoft Visual Studio installed, it returns `None`.

### 14.7.57 tools.apple_dot_clean()

```python
def apple_dot_clean(folder)
```

Remove recursively all `_` files inside `folder`, these files are created by Apple OS when the underlying filesystem cannot store metadata associated to files (they could appear when unzipping a file that has been created in Macos). This tool will remove only the `_` files that are accompanied with a file without that prefix (it will remove `_file.txt` only if `file.txt` exists).

**Parameters:**
• folder (Required): root folder to start deleting .__ files.

14.7.58 tools.Version()

```python
from conans import tools

v = tools.Version("1.2.3-dev23")
assert v < "1.2.3"
```

This is a helper class to work with semantic versions, built on top of `semver.SemVer` class with loose parsing. It exposes all the version components as properties and offers total ordering through compare operators.

Build the `tools.Version` object using any valid string or any object that converts to string, the constructor will raise if the string is not a valid loose semver.

Properties:

- **major**: component `major` of semver version
- **minor**: component `minor` of semver version (defaults to "0")
- **patch**: component `patch` of semver version (defaults to "0")
- **prerelease**: component `prerelease` of semver version (defaults to "")
- **build**: component `build` of semver version (defaults to ""). Take into account that build component doesn’t affect precedence between versions.

14.7.59 tools.to_android_abi()

```python
def to_android_abi(arch):
```

Converts Conan style architecture into Android NDK style architecture.

Parameters:

- **arch** (Required): Arch to perform the conversion. Usually this would be `self.settings.arch`.

14.8 Configuration files

These are the most important configuration files, used to customize conan.

14.8.1 conan.conf

The typical location of the `conan.conf` file is the directory `~/.conan/`:

```
[log]
run_to_output = True    # environment CONAN_LOG_RUN_TO_OUTPUT
run_to_file = False     # environment CONAN_LOG_RUN_TO_FILE
level = 50              # environment CONAN_LOGGING_LEVEL
# trace_file =           # environment CONAN_TRACE_FILE
print_run_commands = False # environment CONAN_PRINT_RUN_COMMANDS

[general]
```

(continues on next page)
default_profile = default
compression_level = 9  # environment CONAN_COMPRESSION_LEVEL
sysrequires_sudo = True  # environment CONAN_SYSREQUIRES_SUDO
request_timeout = 60  # environment CONAN_REQUEST_TIMEOUT (seconds)
default_package_id_mode = semver_direct_mode  # environment CONAN_DEFAULT_PACKAGE_ID_MODE
  # retry = 2  # environment CONAN_RETRY
  # retry_wait = 5  # environment CONAN_RETRY_WAIT (seconds)
  # sysrequires_mode = enabled  # environment CONAN_SYSREQUIRES_MODE (allowed modes enabled/verify/disabled)
  # vs_installation_preference = Enterprise, Professional, Community, BuildTools  #
    # environment CONAN_VS_INSTALLATION_PREFERENCE
  # verbose_traceback = False  # environment CONAN_VERBOSE_TRACEBACK
  # error_on_override = False  # environment CONAN_ERROR_ON_OVERRIDE
  # bash_path = ""  # environment CONAN_BASH_PATH (only windows)
  # recipe_linter = False  # environment CONAN_RECIPE_LINTER
  # read_only_cache = True  # environment CONAN_READ_ONLY_CACHE
  # pylontrc = path/to/pylintrc_file  # environment CONAN_PYLINTRC
  # cache_no_locks = True  # environment CONAN_CACHE_NO_LOCKS
  # user_home_short = your_path  # environment CONAN_USER_HOME_SHORT
  # use_always_short_paths = False  # environment CONAN_USE_ALWAYS_SHORT_PATHS
  # skip_vs_projects_upgrade = False  # environment CONAN_SKIP_VS_PROJECTS_UPGRADE
  # non_interactive = False  # environment CONAN_NON_INTERACTIVE
  # skip_broken_symlinks_check = False  # environment CONAN_SKIP_BROKEN_SYMLINKS_CHECK

# conan_make_program = make  # environment CONAN_MAKE_PROGRAM (overrides the make program used in AutoToolsBuildEnvironment.make)
# conan_cmake_program = cmake  # environment CONAN_CMAKE_PROGRAM (overrides the make program used in CMake.cmake_program)

# cmake_generator  # environment CONAN_CMAKE_GENERATOR
# https://vtk.org/Wiki/CMake_Cross_Compiling
# cmake_generator_platform  # environment CONAN_CMAKE_GENERATOR_PLATFORM
# cmake_toolchain_file  # environment CONAN_CMAKE_TOOLCHAIN_FILE
# cmake_system_name  # environment CONAN_CMAKE_SYSTEM_NAME
# cmake_system_version  # environment CONAN_CMAKE_SYSTEM_VERSION
# cmake_system_processor  # environment CONAN_CMAKE_SYSTEM_PROCESSOR
# cmake_find_root_path  # environment CONAN_CMAKE_FIND_ROOT_PATH
# cmake_find_root_path_mode_program  # environment CONAN_CMAKE_FIND_ROOT_PATH_MODE_PROGRAM
# cmake_find_root_path_mode_library  # environment CONAN_CMAKE_FIND_ROOT_PATH_MODE_LIBRARY
# cmake_find_root_path_mode_include  # environment CONAN_CMAKE_FIND_ROOT_PATH_MODE_INCLUDE

# msbuild_verbosity = minimal  # environment CONAN_MSBUILD_VERBOSITY
# cpu_count = 1  # environment CONAN_CPU_COUNT

# Change the default location for building test packages to a temporary folder which is deleted after the test.
# temp_test_folder = True  # environment CONAN_TEMP_TEST_FOLDER
# cacert_path  # environment CONAN_CACERT_PATH

[storage]
# This is the default path, but you can write your own. It must be an absolute path or a
# (continues on next page)
# path beginning with "~" (if the environment var CONAN_USER_HOME is specified, this_
→ directory, even
# with "~/", will be relative to the conan user home, not to the system user home)
p = ./data

[proxies]
# Empty section will try to use system proxies.
# If don't want proxy at all, remove section [proxies]
# As documented in http://docs.python-requests.org/en/latest/user/advanced/#proxies -
→ but see below
# for proxies to specific hosts
# http = http://user:pass@10.10.1.10:3128/
# http = http://10.10.1.10:3128
# https = http://10.10.1.10:1080
# To specify a proxy for a specific host or hosts, use multiple lines each specifying_
→ host = proxy-spec
# http =
# hostname.to.be.proxied.com = http://user:pass@10.10.1.10:3128
# You can skip the proxy for the matching (fnmatch) urls (comma-separated)
# no_proxy_match = *bintray.com*, https://myserver.*

[hooks] # environment CONAN_HOOKS
attribute_checker

# Default settings now declared in the default profile

Log

The level variable, defaulted to 50 (critical events), declares the LOG level. If you want to show more detailed
logging information, set this variable to lower values, as 10 to show debug information. You can also adjust the
environment variable CONAN_LOGGING_LEVEL.

The print_run_commands, when is 1, Conan will print the executed commands in self.run to the output. You
can also adjust the environment variable CONAN_PRINT_RUN_COMMANDS

The run_to_file variable, defaulted to False, will print the output from the self.run executions to the path that
the variable specifies. You can also adjust the environment variable CONAN_LOG_RUN_TO_FILE.

The run_to_output variable, defaulted to 1, will print to the stdout the output from the self.run executions
in the conanfile. You can also adjust the environment variable CONAN_LOG_RUN_TO_OUTPUT.

The trace_file variable enable extra logging information about your conan command executions. Set it with an
absolute path to a file. You can also adjust the environment variable CONAN_TRACE_FILE.

General

The vs_installation_preference variable determines the preference of usage when searching a Visual
installation. The order of preference by default is Enterprise, Professional, Community and BuildTools. It can
be fixed to just one type of installation like only BuildTools. You can also adjust the environment variable
CONAN_VS_INSTALLATION_PREFERENCE.

The verbose_traceback variable will print the complete traceback when an error occurs in a recipe or even in
the conan code base, allowing to debug the detected error.

The error_on_override turn the messages related to dependencies overriding into errors. When a downstream
package overrides some dependency upstream, if this variable is True then an error will be raised; to bypass these
errors those requirements should be declared explicitly with the `override` keyword.

The `bash_path` variable is used only in windows to help the `tools.run_in_windows_bash()` function to locate our Cygwin/MSYS2 bash. Set it with the bash executable path if it’s not in the PATH or you want to use a different one.

The `cache_no_locks` variable is used to disable locking mechanism of local cache. This is primary used for debugging purposes, and in general it’s not recommended to disable locks otherwise, as it may result in corrupted packages.

The `default_package_id_mode` changes the way package IDs are computed. By default, if not specified it will be `semver_direct_mode`, but can change to any value defined in Using package_id() for Package Dependencies.

The `cmake_***` variables will declare the corresponding CMake variable when you use the `cmake generator` and the `CMake build tool`.

The `msbuild_verbosity` variable is used only by MSBuild and CMake build helpers. For the CMake build helper, it has an effect only for ‘Visual Studio’ generators. Variable defines verbosity level used by the `msbuild` tool, as documented on MSDN <https://docs.microsoft.com/en-us/visualstudio/msbuild/msbuild-command-line-reference?view=vs-2017>. By default, ‘minimal’ verbosity level is used, matching the Visual Studio IDE behavior. Allowed values are (in ascending order): ‘quiet’, ‘minimal’, ‘normal’, ‘detailed’, ‘diagnostic’. You can also adjust the environment variable `CONAN_MSBUILD_VERBOSITY`.

The `conan_make_program` variable used by CMake and AutotoolsBuildEnvironment build helpers. It overrides a default `make` executable, might be useful in case you need to use a different make (e.g. BSD Make instead of GNU Make, or MinGW Make). Set it with the make executable path if it’s not in the PATH or you want to use a different one.

The `conan_cmake_program` variable used only by CMake build helper. It overrides a default `cmake` executable, might be useful in case you need to use a CMake wrapper tool (such as scan build). Set it with the cmake executable path if it’s not in the PATH or you want to use a different one.

The `cpu_count` variable set the number of cores that the `tools.cpu_count()` will return, by default the number of cores available in your machine. Conan recipes can use the `cpu_count()` tool to build the library using more than one core.

The `pylintrc` variable points to a custom `pylintrc` file that allows configuring custom rules for the python linter executed at export time. A use case could be to define some custom indents (though the standard pep8 4-spaces indent is recommended, there are companies that define different styles). The `pylintrc` file has the form:

```
[FORMAT]
indent-string=' ' 
```

Running `pylint --generate-rcfile` will output a complete rcfile with comments explaining the fields.

The `recipe_linter` variable allows to disable the package recipe analysis (linting) executed at `conan install`. Please note that this linting is very recommended, specially for sharing package recipes and collaborating with others.

The `retry` variable allows to set up the global default value for the number of retries in all commands related to download/upload. User can override the value provided by the variable if the command provides an argument with the same name.

The `retry_wait` variable allows to set up the global default value for the time (in seconds) to wait until the next retry on failures in all commands related to download/upload. User can override the value provided by the variable if the command provides an argument with the same name.

The `sysrequires_mode` variable, defaulted to `enabled` (allowed modes `enabled/verify/disabled`) controls whether system packages should be installed into the system via SystemPackageTool helper, typically used in `system_requirements()`. You can also adjust the environment variable `CONAN_SYSREQUIRES_MODE`.

The `sysrequires_sudo` variable, defaulted to True, controls whether sudo is used for installing apt, yum, etc. system packages via SystemPackageTool. You can also adjust the environment variable
The `request_timeout` variable, defaulted to 30 seconds, controls the time after Conan will stop waiting for a response. Timeout is not a time limit on the entire response download; rather, an exception is raised if the server has not issued a response for timeout seconds (more precisely, if no bytes have been received on the underlying socket for timeout seconds). If no timeout is specified explicitly, it do not timeout.

The `user_home_short` specify the base folder to be used with the short paths feature. If not specified, the packages marked as short paths will be stored in the C:\.conan (or the current drive letter).

If the variable is set to “None” will disable the short_paths feature in Windows, for modern Windows that enable long paths at the system level.

The `verbose_traceback` variable will print the complete traceback when an error occurs in a recipe or even in the conan code base, allowing to debug the detected error.

The `cacert_path` variable lets the user specify a custom path to the cacert.pem file to use in requests. You can also adjust this value using the environment variable CONAN_CACERT_PATH.

The `skip_broken_symlinks_check` variable (defaulted to `False`) allows the existence broken symlinks while creating a package.

### Storage

The `storage.path` variable define the path where all the packages will be stored. By default it is `.data`, which is relative to the folder containing this `conan.conf` file, which by default is the `<userhome>/conan` folder. It can start with “~”, and that will be expanded to the current user home folder. If the environment var CONAN_USER_HOME is specified, the “~” will be replaced by the current Conan home (the folder pointed by the CONAN_USER_HOME environment variable).

On Windows:

- It is recommended to assign it to some unit, e.g. map it to X: in order to avoid hitting the 260 chars path name length limit).
- Also see the short_paths docs to know more about how to mitigate the limitation of 260 chars path name length limit.
- It is recommended to disable the Windows indexer or exclude the storage path to avoid problems (busy resources).

**Note:** If you want to change the default “conan home” (directory where `conan.conf` file is) you can adjust the environment variable CONAN_USER_HOME.

### Proxies

If you are not using proxies at all, or you want to use the proxies specified by the operating system, just remove the [proxies] section completely. You can run `conan config rm proxies`.

If you leave the [proxies] section blank, conan will copy the system configured proxies, but if you configured some exclusion rule it won’t work:

```
[proxies]
# Empty section will try to use system proxies.
# If you don't want Conan to mess with proxies at all, remove section [proxies]
```
You can specify http and https proxies as follows. Use the `no_proxy_match` keyword to specify a list of URLs or patterns that will skip the proxy:

```
[proxies]
# As documented in http://docs.python-requests.org/en/latest/user/advanced/#proxies
http: http://user:pass@10.10.1.10:3128/
http: http://10.10.1.10:3128
https: http://10.10.1.10:1080
http: http://10.10.2.10
  hostname1.to.be.proxied.com = http://user:pass@10.10.3.10
  hostname2.to.be.proxied.com = http://user:pass@10.10.4.10
```

Use `http=None` and/or `https=None` to disable the usage of a proxy.

To nominate a proxy for a specific scheme and host only, add `host.to.proxy=` in front of the url of the proxy (the `host.to.proxy` name must exactly match the host name that should be proxied). You can list several `host name = proxy` pairs on separate indented lines.

You can still specify a default proxy, without a host, which will be used if none of the host names match. If you do not, then the proxy is disabled for non-matching hosts.

If this fails, you might also try to set environment variables:

```
# linux/osx
$ export HTTP_PROXY="http://10.10.1.10:3128"
$ export HTTPS_PROXY="http://10.10.1.10:1080"

# with user/password
$ export HTTP_PROXY="http://user:pass@10.10.1.10:3128/"
$ export HTTPS_PROXY="http://user:pass@10.10.1.10:3128/"

# windows (note, no quotes here)
$ set HTTP_PROXY=http://10.10.1.10:3128
$ set HTTPS_PROXY=http://10.10.1.10:1080
```

### 14.8.2 profiles/default

This is the typical `~/.conan/profiles/default` file:

```
[build_requires]
[settings]
  os=Macos
  arch=x86_64
  compiler=apple-clang
  compiler.version=8.1
  compiler.libcxx=libc++
  build_type=Release

[options]
[env]
```

The settings defaults are the setting values used whenever you issue a `conan install` command over a `conanfile` in one of your projects. The initial values for these default settings are auto-detected the first time you run a `conan` command.

You can override the default settings using the `-s` parameter in `conan install` and `conan info` commands but when you specify a profile, `conan install --profile gcc48`, the default profile won’t be applied, unless you specify it with an `include()` statement.
Listing 16: my_clang_profile

```yaml
include(default)

[settings]
compiler=clang
compiler.version=3.5
compiler.libcxx=libstdc++11

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

**Tip:** Default profile can be overridden using the environment variable CONAN_DEFAULT_PROFILE_PATH.

**See also:**

Check the section *Profiles* to read more about this feature.

### 14.8.3 settings.yml

The input settings for packages in Conan are predefined in ~/.conan/settings.yml file, so only a few like os or compiler are possible. These are the default values, but it is possible to customize them, see Customizing settings.

```yaml
# Only for cross building, 'os_build/arch_build' is the system that runs Conan
os_build: [Windows, WindowsStore, Linux, Macos, FreeBSD, SunOS, AIX]
arch_build: [x86, x86_64, ppc32be, ppc32, ppc64le, ppc64, armv5e, armv5hf, armv6,
    →armv7, armv7hf, armv7s, armv7k, armv8, armv8_32, armv8.3, sparc, sparcv9, mips,
    →mips64, avr, s390, s390x, sh4le]

# Only for building cross compilation tools, 'os_target/arch_target' is the system for which the tools generate code
os_target: [Windows, Linux, Macos, Android, iOS, watchOS, tvOS, FreeBSD, SunOS, AIX,
    →Arduinos, Neutrino]
arch_target: [x86, x86_64, ppc32be, ppc32, ppc64le, ppc64, armv5e, armv5hf, armv6,
    →armv7, armv7hf, armv7s, armv7k, armv8, armv8_32, armv8.3, sparc, sparcv9, mips,
    →mips64, avr, s390, s390x, asm.js, wasm, sh4le]

# Rest of the settings are "host" settings:
# - For native building/cross building: Where the library/program will run.
# - For building cross compilation tools: Where the cross compiler will run.
```

(continues on next page)
### Android

- `api_level`: ANY

### iOS

- `version`: "7.0", "7.1", "8.0", "8.1", "8.2", "8.3", "9.0", "9.1", "9.2", "9.3", ...
- `watchOS`
  - `version`: "4.0", "4.1", "4.2", "4.3", "5.0", "5.1"

### tvOS

- `version`: "11.0", "11.1", "11.2", "11.3", "11.4", "12.0", "12.1"

### FreeBSD

### SunOS

### AIX

- `board`: ANY

### Arduino

- `board`: ANY

### Emscripten

- `version`: "6.4", "6.5", "6.6", "7.0"

### Neutrino

- `version`: "6.4", "6.5", "6.6", "7.0"

### Arch

- `arch`: [x86, x86_64, ppc32be, ppc32, ppc64le, ppc64, armv4, armv4i, armv5el, armv5hf, ...
- `compiler`
  - `sun-cc`
    - `version`: "5.10", "5.11", "5.12", "5.13", "5.14"
    - `threads`: [None, posix]

### GCC

- `version`: "4.1", "4.4", "4.5", "4.6", "4.7", "4.8", "4.9", ...

### Libcxx

- `libcxx`: [libCstd, libstdcxx, libstlport, libstdc++]

### Clang

- `version`: "3.3", "3.4", "3.5", "3.6", "3.7", "3.8", "3.9", "4.0", ...

### Apple-Clang

- `version`: "5.0", "5.1", "6.0", "6.1", "7.0", "7.3", "8.0", "8.1", "9.0", "9.1", ...

### Navigator

- `version`: "8", "9", "10", "11", "12", "14", "15", "16"

### Toolset


### Cppstd

- `cppstd`: [None, 98, gnu98, 11, gnu11, 14, gnu14, 17, gnu17, 20, gnu20]

### Visual Studio

- `runtime`: [MD, MT, MTd, MDd]

### Exception

- `exception`: [None, dwarf2, sjlj, seh] # Windows MinGW

### Cppstd

- `cppstd`: [None, 98, gnu98, 11, gnu11, 14, gnu14, 17, gnu17, 20, gnu20]
As you can see, the possible values settings can take are restricted 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. 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 [None, ANY].

However, this configuration file can be you can modified to any needs, including new settings or subsettings and their values. If you want to distribute a unified settings.yml file you can use the conan config install command.

Note: The settings.yml file is not perfect nor definitive and surely incomplete. Please share any suggestion in the Conan issue tracker with any missing settings and values that could make sense for other users.

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.

### 14.8.4 registry.txt / registry.json

**Note:** *registry.json* was introduced in Conan 1.9 and it substitutes the *registry.txt*. Both files are equivalent in content.

This file is generally automatically managed and it stores information about the remotes configured in the Conan client and the installed packages in your cache associated with the remote they were retrieved from.

**Important:** The recommendation is not to modify this file directly and only use it as registry information. In case you want to change a remote the information of the remotes can be accessed or changed via the `conan remote` command.

**Listing 17: registry.txt**

```
conan-center https://conan.bintray.com True
local http://localhost:9300 True
Hello/0.1@demo/testing local
```

The first section of the file is listing `remote-name: remote-url verify_ssl`. Adding, removing or changing those lines, will add, remove or change the respective remote. If `verify_ssl` is enabled, Conan will verify the SSL certificates for that remote server.

The second part of the file contains a list of package references and the remote-name. This is a reference to which remote was that package retrieved from, which will act also as the default for operations on that package.

Here you have an example of the file *registry.json*:

**Listing 18: registry.json**

```
{
   "remotes": [
      {
      }
```

(continues on next page)
"name":"conan-center",
"url":"https://api.bintray.com/conan/conan/conan-center",
"verify_ssl":true
},
{
"name":"artifactory_local",
"url":"http://192.168.43.191:8081/artifactory/api/conan/conan-local",
"verify_ssl":true
},
{
"name":"bincrafters",
"url":"https://api.bintray.com/conan/bincrafters/public-conan",
"verify_ssl":true
}
],
"references":{
"nanomsg/1.1.2@bincrafters/stable":"bincrafters",
"Poco/1.9.0@pocoproject/stable:09378ed7f51185386e9f04b212b79fe2d12d005c":
→"conan-center",
"hello/1.0@user/channel:2bb76c9adac7b8cd7c5e3b377ac9f06934aba606":
→"artifactory_local"
},
"package_references":{
"nanomsg/1.1.2@bincrafters/stable:26d575619895d584ff4fb07701901d53ff4cdd6b":
→"bincrafters",
"Poco/1.9.0@pocoproject/stable:09378ed7f51185386e9f04b212b79fe2d12d005c":
→"conan-center",
"hello/1.0@user/channel:2bb76c9adac7b8cd7c5e3b377ac9f06934aba606":
→"artifactory_local"
}
}

14.8.5 client.crt / client.key

Conan support client TLS certificates. Create a client.crt with the client certificate in the conan home directory (default ~/.conan) and a client.key with the private key.

You could also create only the client.crt file containing both the certificate and the private key concatenated.

14.8.6 artifacts.properties

This file is used to send custom headers in the PUT requests that conan upload command does:

```
.conan/artifacts.properties
```

custom_header1=Value1

custom_header2=45

Artifactory users can use this file to set file properties for the uploaded files. The variables should have the prefix artifact_property. You can use ; to set multiple values to a property:

```
.conan/artifacts.properties
```

14.8.7 Editable layout files

This file contain information consumed by editable packages. It is an .ini file listing the directories that Conan should use for the packages that are opened in editable mode. Before parsing this file Conan runs Jinja2 template engine with the settings, options and reference objects, so you can add any logic to this files:

```
# Affects to all packages but cool/version@user/dev
[includedirs]
src/include

# using placeholders from conan settings and options
[libdirs]
build/{{settings.build_type}}/{{settings.arch}}

[bindirs]
{% if options.shared %}
built/{{settings.build_type}}/shared
{% else %}
built/{{settings.build_type}}/static
{% endif %}

# Affects only to cool/version@user/dev
[cool/version@user/dev:includedirs]
src/core/include
src/cmp_a/include

# The source_folder, build_folder are useful for workspaces
[source_folder]
src

[build_folder]
built/{{settings.build_type}}/{{settings.arch}}
```

The specific sections using a package reference will have higher priority than the general ones.

This file can live in the conan cache, in the .conan/layouts folder, or in a user folder, like inside the source repo.

If there exists a .conan/layouts/default layout file in the cache and no layout file is specified in the conan editable add <path> <reference> command, that file will be used.

The [source_folder] and [build_folder] are useful for workspaces. For example, when using cmake workspace-generator, it will locate the CMakeLists.txt of each package in editable mode in the [source_folder] and it will use the [build_folder] as the base folder for the build temporary files.

It is possible to define out-of-source builds for workspaces, using relative paths and the reference argument. The following could be used to locate the build artifacts of an editable package in a sibling build/<package-name> folder:

```
[build_folder]
./build/{{reference.name}}/{{settings.build_type}}
```

(continues on next page)
14.9 Environment variables

These are the environment variables used to customize Conan.

Most of them can be set in the `conan.conf` configuration file (inside your `<userhome>/ .conan` folder). However, this environment variables will take precedence over the `conan.conf` configuration.

### 14.9.1 CMAKE RELATED VARIABLES

There are some Conan environment variables that will set the equivalent CMake variable using the `cmake generator` and the `CMake build tool`:

<table>
<thead>
<tr>
<th>Variable</th>
<th>CMake set variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONAN_CMAKE_TOOLCHAIN_FILE</td>
<td>CMAKE_TOOLCHAIN_FILE</td>
</tr>
<tr>
<td>CONAN_CMAKE_SYSTEM_NAME</td>
<td>CMAKE_SYSTEM_NAME</td>
</tr>
<tr>
<td>CONAN_CMAKE_SYSTEM_VERSION</td>
<td>CMAKE_SYSTEM_VERSION</td>
</tr>
<tr>
<td>CONAN_CMAKE_SYSTEM_PROCESSOR</td>
<td>CMAKE_SYSTEM_PROCESSOR</td>
</tr>
<tr>
<td>CONAN_CMAKE_FIND_ROOT_PATH</td>
<td>CMAKE_FIND_ROOT_PATH</td>
</tr>
<tr>
<td>CONAN_CMAKE_FIND_ROOT_PATH_MODE_PROGRAM</td>
<td>CMAKE_FIND_ROOT_PATH_MODE_PROGRAM</td>
</tr>
<tr>
<td>CONAN_CMAKE_FIND_ROOT_PATH_MODE_LIBRARY</td>
<td>CMAKE_FIND_ROOT_PATH_MODE_LIBRARY</td>
</tr>
<tr>
<td>CONAN_CMAKE_FIND_ROOT_PATH_MODE_INCLUDE</td>
<td>CMAKE_FIND_ROOT_PATH_MODE_INCLUDE</td>
</tr>
<tr>
<td>CONAN_CMAKE_GENERATOR_PLATFORM</td>
<td>CMAKE_GENERATOR_PLATFORM</td>
</tr>
<tr>
<td>CONAN_CMAKE_ANDROID_NDK</td>
<td>CMAKE_ANDROID_NDK</td>
</tr>
</tbody>
</table>

See also:

See CMake cross building wiki

### 14.9.2 CONAN_BASH_PATH

**Defaulted to:** Not defined

Used only in windows to help the `tools.run_in_windows_bash()` function to locate our Cygwin/MSYS2 bash. Set it with the bash executable path if it’s not in the `PATH` or you want to use a different one.

### 14.9.3 CONAN_CACHE_NO_LOCKS

**Defaulted to:** False/0

Set it to True/1 to disable locking mechanism of local cache. Set it to False/0 to enable locking mechanism of local cache. Use it with caution, and only for debugging purposes. Disabling locks may easily lead to corrupted...
packages. Not recommended for production environments, and in general should be used for conan development and contributions only.

14.9.4 CONAN_CMAKE_GENERATOR

Conan CMake helper class is just a convenience to help to translate Conan settings and options into CMake parameters, but you can easily do it yourself, or adapt it.

For some compiler configurations, as `gcc` it will use by default the `Unix Makefiles` CMake generator. Note that this is not a package settings, building it with makefiles or other build system, as Ninja, should lead to the same binary if using appropriately the same underlying compiler settings. So it doesn’t make sense to provide a setting or option for this.

So it can be set with the environment variable `CONAN_CMAKE_GENERATOR`. Just set its value to your desired CMake generator (as `Ninja`).

14.9.5 CONAN_CMAKE_GENERATOR_PLATFORM

Defines generator platform to be used by particular CMake generator (see `CMAKE_GENERATOR_PLATFORM` documentation <https://cmake.org/cmake/help/latest/variable/CMAKE_GENERATOR_PLATFORM.html>). Resulting value is passed to the `cmake` command line (–A argument) by the Conan CMake helper class during the configuration step. Passing `None` causes auto-detection, which currently only happens for the Visual Studio 16 2019 generator. The detection is according to the following table:

<table>
<thead>
<tr>
<th>settings.arch</th>
<th>generator platform</th>
</tr>
</thead>
<tbody>
<tr>
<td>x86</td>
<td>Win32</td>
</tr>
<tr>
<td>x86_64</td>
<td>x64</td>
</tr>
<tr>
<td>armv7</td>
<td>ARM</td>
</tr>
<tr>
<td>armv8</td>
<td>ARM64</td>
</tr>
<tr>
<td>other</td>
<td>(none)</td>
</tr>
</tbody>
</table>

For any other generators besides the Visual Studio 16 2019 generator, detection results in no generator platform applied (and no –A argument passed to the CMake command line).

14.9.6 CONAN_COLOR_DARK

Defaulted to: `False/0`

Set it to `True/1` to use dark colors in the terminal output, instead of light ones. Useful for terminal or consoles with light colors as white, so text is rendered in Blue, Black, Magenta, instead of Yellow, Cyan, White.

14.9.7 CONAN_COLOR_DISPLAY

Defaulted to: Not defined

By default if undefined Conan output will use color if a tty is detected.

Set it to `False/0` to remove console output colors. Set it to `True/1` to force console output colors.
14.9.8 CONAN_COMPRESSION_LEVEL

Defaulted to: 9

Conan uses .tgz compression for archives before uploading them to remotes. The default compression level is good and fast enough for most cases, but users with huge packages might want to change it and set CONAN_COMPRESSION_LEVEL environment variable to a lower number, which is able to get slightly bigger archives but much better compression speed.

14.9.9 CONAN_CPU_COUNT

Defaulted to: Number of available cores in your machine.

Set the number of cores that the tools.cpu_count() will return. Conan recipes can use the cpu_count() tool to build the library using more than one core.

14.9.10 CONAN_DEFAULT_PROFILE_PATH

Defaulted to: Not defined

This variable can be used to define a path to an existing profile file that Conan will use as default. If relative, the path will be resolved from the profiles folder.

14.9.11 CONAN_NON_INTERACTIVE

Defaulted to: False/0

This environment variable, if set to True/1, will prevent interactive prompts. Invocations of Conan commands where an interactive prompt would otherwise appear, will fail instead.

This variable can also be set in conan.conf as non_interactive = True in the [general] section.

14.9.12 CONAN_ENV_XXXX_YYYY

You can override the default settings (located in your ~/.conan/profiles/default directory) with environment variables.

The XXXX is the setting name upper-case, and the YYYY (optional) is the sub-setting name.

Examples:

- Override the default compiler:

  CONAN_ENV_COMPILER = "Visual Studio"

- Override the default compiler version:

  CONAN_ENV_COMPILER_VERSION = "14"

- Override the architecture:

  CONAN_ENV_ARCH = "x86"
14.9.13 CONAN_LOG_RUN_TO_FILE

Defaulted to: 0

If set to 1 will log every `self.run("(Some command)")` command output in a file called `conan_run.log`. That file will be located in the current execution directory, so if we call `self.run` in the conanfile.py's build method, the file will be located in the build folder.

In case we execute `self.run` in our `source()` method, the `conan_run.log` will be created in the source directory, but then conan will copy it to the build folder following the regular execution flow. So the `conan_run.log` will contain all the logs from your conanfile.py command executions.

The file can be included in the Conan package (for debugging purposes) using the `package` method.

```python
def package(self):
    self.copy(pattern="conan_run.log", dst='', keep_path=False)
```

14.9.14 CONAN_LOG_RUN_TO_OUTPUT

Defaulted to: 1

If set to 0 Conan won’t print the command output to the stdout. Can be used with `CONAN_LOG_RUN_TO_FILE` set to 1 to log only to file and not printing the output.

14.9.15 CONAN_LOGGING_LEVEL

Defaulted to: 50

By default Conan logging level is only set for critical events. If you want to show more detailed logging information, set this variable to lower values, as 10 to show debug information.

14.9.16 CONAN_LOGIN_USERNAME, CONAN_LOGIN_USERNAME_{REMOTE_NAME}

Defaulted to: Not defined

You can define the username for the authentication process using environment variables. Conan will use a variable `CONAN_LOGIN_USERNAME_{REMOTE_NAME}`, if the variable is not declared Conan will use the variable `CONAN_LOGIN_USERNAME`, if the variable is not declared either, Conan will request to the user to input a username.

These variables are useful for unattended executions like CI servers or automated tasks.

If the remote name contains “-” you have to replace it with “_” in the variable name:

For example: For a remote named “conan-center”:

```
SET CONAN_LOGIN_USERNAME_CONAN_CENTER=MyUser
```

14.9.17 CONAN_MAKE_PROGRAM

Defaulted to: Not defined

Specify an alternative `make` program to use with:

- The build helper `AutoToolsBuildEnvironment`. Will invoke the specified executable in the `make` method.
• The build helper build helper CMake. By adjusting the CMake variable CMAKE_MAKE_PROGRAM.

For example:

```
CONAN_MAKE_PROGRAM="/path/to/mingw32-make"
# Or only the exe name if it is in the path
CONAN_MAKE_PROGRAM="mingw32-make"
```

14.9.18 CONAN_CMAKE_PROGRAM

**Defaulted to:** Not defined

Specify an alternative *cmake* program to use with *CMake* build helper.

For example:

```
CONAN_CMAKE_PROGRAM="scan-build cmake"
```

14.9.19 CONAN_MSBUILD_VERBOSITY

**Defaulted to:** Not defined

Specify `MSBuild` verbosity level to use with:

• The build helper *CMake*.
• The build helper *MSBuild*.

For list of allowed values and their meaning, check out the MSBuild documentation.

14.9.20 CONAN_PASSWORD, CONAN_PASSWORD_{REMOTE_NAME}

**Defaulted to:** Not defined

You can define the authentication password using environment variables. Conan will use a variable CO-
NAN_PASSWORD_{REMOTE_NAME}, if the variable is not declared Conan will use the variable CO-
NAN_PASSWORD, if the variable is not declared either, Conan will request to the user to input a password.

These variables are useful for unattended executions like CI servers or automated tasks.

The remote name is transformed to all uppercase. If the remote name contains “-“, you have to replace it with “_” in
the variable name.

For example, for a remote named “conan-center”:

```
SET CONAN_PASSWORD_CONAN_CENTER=Mypassword
```

14.9.21 CONAN_HOOKS

**Defaulted to:** Not defined

Can be set to a comma separated list with the names of the hooks that will be executed when running a Conan
command.
14.9.22 CONAN_PRINT_RUN_COMMANDS

Defaulted to: 0

If set to 1, every `self.run("{Some command}\")` call will log the executed command `{Some command}` to the output.

For example: In the `conanfile.py` file:

```python
self.run("cd %s && %s ./configure" % (self.ZIP_FOLDER_NAME, env_line))
```

Will print to the output (stout and/or file):

```
--Running------
> cd zlib-1.2.9 && env LIBS="" LDFLAGS="-m64 $LDFLAGS" CFLAGS="-mstackrealign -fPIC $CFLAGS -m64 -s -DNDEBUG " CPPFLAGS="$CPPFLAGS -m64 -s -DNDEBUG " C_INCLUDE_PATH=$C_INCLUDE_PATH: CPLUS_INCLUDE_PATH=$CPLUS_INCLUDE_PATH: ./configure
----------------
...```

14.9.23 CONAN_READ_ONLY_CACHE

Defaulted to: Not defined

This environment variable if defined, will make the Conan cache read-only. This could prevent developers to accidentally edit some header of their dependencies while navigating code in their IDEs.

This variable can also be set in `conan.conf` as `read_only_cache = True` in the `[general]` section.

The packages are made read-only in two points: when a package is built from sources, and when a package is retrieved from a remote repository.

The packages are not modified for upload, so users should take that into consideration before uploading packages, as they will be read-only and that could have other side-effects.

**Warning:** It is not recommended to upload packages directly from developers machines with read-only mode as it could lead to inconsistencies. For better reproducibility we recommend that packages are created and uploaded by CI machines.

14.9.24 CONAN_RUN_TESTS

Defaulted to: Not defined (True/False if defined)

This environment variable (if defined) can be used in `conanfile.py` to enable/disable the tests for a library or application.

It can be used as a convention variable and it’s specially useful if a library has unit tests and you are doing *cross building*, the target binary can’t be executed in current host machine building the package.

It can be defined in your profile files at `~/.conan/profiles`

```bash
...[env]
CONAN_RUN_TESTS=False
```

or declared in command line when invoking `conan install` to reduce the variable scope for conan execution.
$$\text{conan install . -e CONAN_RUN_TEST=0}$$

See how to retrieve the value with `tools.get_env()` and check a use case with a header only with unit tests recipe while cross building.

See example of build method in `conanfile.py` to enable/disable running tests with CMake:

```python
from conans import ConanFile, CMake, tools

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

    def build(self):
        cmake = CMake(self)
        cmake.configure()
        cmake.build()
        if tools.get_env("CONAN_RUN_TESTS", True):
            cmake.test()
```

### 14.9.25 CONAN_SKIP_VS_PROJECTS_UPGRADE

**Defaulted to:** False/0

When set to True/1, the `tools.build_sln_command()` [DEPRECATED], the `tools.msvc_build_command()` [DEPRECATED] and the `MSBuild()` build helper, will not call `devenv` command to upgrade the `sln` project, irrespective of the `upgrade_project` parameter value.

### 14.9.26 CONAN_SYSREQUIRES_MODE

**Defaulted to:** enabled allowed values enabled/verify/disabled

This environment variable controls whether system packages should be installed into the system via `SystemPackageTool` helper, typically used in `system_requirements()`.

See values behavior:

- **enabled**: Default value and any call to install method of `SystemPackageTool` helper should modify the system packages.
- **verify**: Display a report of system packages to be installed and abort with exception. Useful if you don’t want to allow Conan to modify your system but you want to get a report of packages to be installed.
- **disabled**: Display a report of system packages that should be installed but continue the Conan execution and doesn’t install any package in your system. Useful if you want to keep manual control of these dependencies, for example in your development environment.

### 14.9.27 CONAN_SYSREQUIRES_SUDO

**Defaulted to:** True/1

This environment variable controls whether `sudo` is used for installing apt, yum, etc. system packages via `SystemPackageTool` helper, typically used in `system_requirements()`. By default when the environment variable does not exist, “True” is assumed, and `sudo` is automatically prefixed in front of package management commands. If you set this to “False” or “0” `sudo` will not be prefixed in front of the commands, however installation or updates of some packages may fail due to a lack of privilege, depending on the user account Conan is running under.
14.9.28 CONAN_TEMP_TEST_FOLDER

Defaulted to: False/0

Activating this variable will make build folder of test_package to be created in the temporary folder of your machine.

14.9.29 CONAN_TRACE_FILE

Defaulted to: Not defined

If you want extra logging information about your Conan command executions, you can enable it by setting the CONAN_TRACE_FILE environment variable. Set it with an absolute path to a file.

```bash
export CONAN_TRACE_FILE=/tmp/conan_trace.log
```

When the Conan command is executed, some traces will be appended to the specified file. Each line contains a JSON object. The _action field contains the action type, like COMMAND for command executions, EXCEPTION for errors and REST_API_CALL for HTTP calls to a remote.

The logger will append the traces until the CONAN_TRACE_FILE variable is unset or pointed to a different file.

See also:
Read more here: How to log and debug a conan execution

14.9.30 CONAN_USERNAME, CONAN_CHANNEL

These environment variables will be checked when using self.user or self.channel in package recipes in user space, where the user and channel have not been assigned yet (they are assigned when exported in the local cache).

See also:
Read more about it in user, channel

14.9.31 CONAN_USER_HOME

Defaulted to: Not defined

Allows defining a custom Conan cache directory. Can be useful for concurrent builds under different users in CI, to retrieve and store per-project specific dependencies (useful for deployment, for example).

See also:
Read more about it in Conan local cache: concurrency, Continuous Integration, isolation

14.9.32 CONAN_USER_HOME_SHORT

Defaulted to: Not defined

Specify the base folder to be used with the short_paths feature. When not specified, the packages marked as short_paths will be stored in the C:\.conan (or the current drive letter).

If set to None, it will disable the short_paths feature in Windows for modern Windows that enable long paths at the system level.

14.9. Environment variables
14.9.33 CONAN_USE_ALWAYS_SHORT_PATHS

Defaulted to: Not defined

If defined to True or 1, every package will be stored in the *short paths directory* resolved by Conan after evaluating CONAN_USER_HOME_SHORT variable (see above). This variable, therefore, overrides the value defined in recipes for the attribute *short paths*.

14.9.34 CONAN_VERBOSE_TRACEBACK

Defaulted to: 0

When an error is raised in a recipe or even in the Conan code base, if set to 1 it will show the complete traceback to ease the debugging.

14.9.35 CONAN_ERROR_ON_OVERRIDE

**Defaulted to**: False

When a consumer overrides one transitive requirement without using explicitly the keyword override Conan will raise an error if this environment variable is set to True.

This variable can also be set in the *conan.conf* file under the section [general].

14.9.36 CONAN_VS_INSTALLATION_PREFERENCE

Defaulted to: Enterprise, Professional, Community, BuildTools

This environment variable defines the order of preference when searching for a Visual installation product. This would affect every tool that uses tools.vs_installation_path() and will search in the order indicated.

For example:

```
set CONAN_VS_INSTALLATION_PREFERENCE=Enterprise, Professional, Community, BuildTools
```

It can also be used to fix the type of installation you want to use indicating just one product type:

```
set CONAN_VS_INSTALLATION_PREFERENCE=BuildTools
```

14.9.37 CONAN_CACERT_PATH

Defaulted to: Not defined

Specify an alternative path to a *cacert.pem* file to be used for requests. This variable overrides the value defined in the conan.conf as cacert_path = <path/to/cacert.pem> under the section [general].

14.9.38 CONAN_DEFAULT_PACKAGE_ID_MODE

Defaulted to: semver_direct_mode

It changes the way package IDs are computed, but can change to any value defined in *Using package_id() for Package Dependencies*. 
14.9.39 CONAN_SKIP_BROKEN_SYMLINKS_CHECK

Defaulted to: False

When set to True/1, Conan will allow the existence broken symlinks while creating a package.

14.10 Hooks [EXPERIMENTAL]

**Warning:** This is an experimental feature subject to breaking changes in future releases.

The Conan hooks are Python functions that are intended to extend the Conan functionalities and let users customize the client behavior at determined execution points.

14.10.1 Hook interface

Here you can see a complete example of all the hook functions available and the different parameters for each of them depending on the context:

```python
def pre_export(output, conanfile, conanfile_path, reference, **kwargs):
    assert conanfile
    output.info("conanfile_path=%s" % conanfile_path)
    output.info("reference=%s" % str(reference))

def post_export(output, conanfile, conanfile_path, reference, **kwargs):
    assert conanfile
    output.info("conanfile_path=%s" % conanfile_path)
    output.info("reference=%s" % str(reference))

def pre_source(output, conanfile, conanfile_path, **kwargs):
    assert conanfile
    if conanfile.in_local_cache:
        output.info("reference=%s" % str(kwargs["reference")))

def post_source(output, conanfile, conanfile_path, **kwargs):
    assert conanfile
    if conanfile.in_local_cache:
        output.info("reference=%s" % str(kwargs["reference")))

def pre_build(output, conanfile, **kwargs):
    assert conanfile
    if conanfile.in_local_cache:
        output.info("reference=%s" % str(kwargs["reference")))
        output.info("package_id=%s" % kwargs["package_id"])
    else:
        output.info("conanfile_path=%s" % kwargs["conanfile_path"])

def post_build(output, conanfile, **kwargs):
    assert conanfile
    if conanfile.in_local_cache:
        output.info("reference=%s" % str(kwargs["reference")))
```

(continues on next page)
def pre_package(output, conanfile, conanfile_path, **kwargs):
    assert conanfile
    output.info("conanfile_path=%s" % conanfile_path)
    if conanfile in local_cache:
        output.info("reference=%s" % str(kwargs["reference"]))
        output.info("package_id=%s" % kwargs["package_id"])

def post_package(output, conanfile, conanfile_path, **kwargs):
    assert conanfile
    output.info("conanfile_path=%s" % conanfile_path)
    if conanfile in local_cache:
        output.info("reference=%s" % str(kwargs["reference"]))
        output.info("package_id=%s" % kwargs["package_id"])

def pre_upload(output, conanfile_path, reference, remote, **kwargs):
    output.info("conanfile_path=%s" % conanfile_path)
    output.info("reference=%s" % str(reference))
    output.info("remote.name=%s" % remote.name)

def post_upload(output, conanfile_path, reference, remote, **kwargs):
    output.info("conanfile_path=%s" % conanfile_path)
    output.info("reference=%s" % str(reference))
    output.info("remote.name=%s" % remote.name)

def pre_upload_recipe(output, conanfile_path, reference, remote, **kwargs):
    output.info("conanfile_path=%s" % conanfile_path)
    output.info("reference=%s" % str(reference))
    output.info("remote.name=%s" % remote.name)

def post_upload_recipe(output, conanfile_path, reference, remote, **kwargs):
    output.info("conanfile_path=%s" % conanfile_path)
    output.info("reference=%s" % str(reference))
    output.info("remote.name=%s" % remote.name)

def pre_upload_package(output, conanfile_path, reference, package_id, remote, **kwargs):
    output.info("conanfile_path=%s" % conanfile_path)
    output.info("reference=%s" % str(reference))
    output.info("package_id=%s" % package_id)
    output.info("remote.name=%s" % remote.name)

def post_upload_package(output, conanfile_path, reference, package_id, remote, **kwargs):
    output.info("conanfile_path=%s" % conanfile_path)
    output.info("reference=%s" % str(reference))
    output.info("package_id=%s" % package_id)
    output.info("remote.name=%s" % remote.name)

def pre_download(output, reference, remote, **kwargs):
    output.info("reference=%s" % str(reference))
    output.info("remote.name=%s" % remote.name)

def post_download(output, conanfile_path, reference, remote, **kwargs):
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.

For download/upload functions, the pre_download() / pre_upload() function is executed first in an conan download / conan upload command. Then pre and post download_recipe() / upload_recipe() and its subsequent pre/post download_package() / upload_package() if that is the case. Finally the general post_download() / post_upload() function is called to wrap up the whole execution.

Important: Pre and post download_recipe() / download_package() are also executed when installing new recipes/packages from remotes using conan create or conan install.

14.10.2 Function parameters

Here you can find the description for each parameter:

- **output**: Output object to print formatted messages during execution with the name of the hook and the function executed, e.g., [HOOK - complete_hook] post_download_package(): This is the remote name: default.
• **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...

• **conanfile_path**: Path to the conanfile.py file whether it is in local cache or in user space.

• **reference**: Named tuple with attributes name, version, user, and channel. Its representation will be a reference like: box2d/2.1.0@user/channel

• **package_id**: String with the computed package ID.

• **remote**: Named tuple with attributes name, url and verify_ssl.

| Availability of parameters for each Hook function depending on the context | Hook Functions* |
|---|---|---|---|---|---|---|---|---|---|
| export() | source() | build() | package() | upload() | download() | upload_recipe() | upload_package() |
| conanfile | Yes | Yes | Yes | Yes | No | post | Yes |
| conanfile_path | pre / post | Yes | user space | pre / post | Yes | post | No |
| reference | Yes | cache | cache | cache | Yes | Yes | Yes |
| package_id | No | No | cache | Yes | Yes | Yes | No |
| remote | No | No | No | No | Yes | Yes | No |

*Hook functions are indicated without pre and post prefixes for simplicity.

**Table legend:**

- **Yes**: Availability in pre and post functions in any context.
- **No**: Not available.
- **pre / post**: Availability in both pre and post functions with different values. e.g. conanfile_path pointing to user space in pre and to local cache in post.
- **post**: Only available in post function.
- **cache**: Only available when the context of the command executed is the local cache. e.g. conan create, conan install...
- **user space**: Only available when the context of the command executed is the user space. e.g. conan build

**Note**: Path to the different folders of the Conan execution flow may be accessible as usual through the conanfile object. See source_folder to learn more.

Some of this parameters does not appear in the signature of the function as they may not be always available (Mostly depending on the recipe living in the local cache or in user space). However, they can be checked with the kwargs parameter.
**Important:** Hook functions should have a `**kwargs` parameter to keep compatibility of new parameters that may be introduced in future versions of Conan.
VIDEOS AND LINKS

- Meeting Embedded 2018: “Continuous Integration of C/C++ for embedded and IoT with Jenkins, Docker and Conan” by Diego Rodriguez-Losada and Daniel Manzaneque.
- CppCon 2018: “Git, CMake, Conan - How to ship and reuse our C++ projects” by Mateusz Pusz.
- JFrog swampUP 2018: “Managing dependencies and toolchains with Conan and Artifactory” by Tobias Hieta
- JFrog swampUP 2018: “Cross building... It’s almost too easy!” by Théo Delrieu.
- JFrog Conan Playlist: “Conan - The C/C++ Package Manager”
- FOSDEM 2018: “Packaging C/C++ libraries with Conan” by Théo Delrieu.
  Includes AndroidNDK package and cross build to Android
- CppCon 2017: “Faster Delivery of Large C/C++ Projects with Conan Package Manager and Efficient Continuous Integration” by Diego Rodriguez-Losada.
- “Conan.io C++ Package Manager demo with SFML” by Charl Botha

Do you have a video, tutorial, blog post that could be useful for other users and would like to share? Please tell us about it or directly send a PR to our docs: https://github.com/conan-io/docs, and we will link it here.
See also:

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

16.1 Upgrading to Conan 1.0

If you have been using a 0.X version of Conan, there are some things to consider when upgrading to version 1.0. These are reflected in the changelog. This section summarizes the most important considerations:

16.1.1 Command line changes

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

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

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

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

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

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

- Scopes were completely removed in Conan 0.30.X
- `self.conanfile_directory` has been removed. Use `self.source_folder`, `self.build_folder`, etc. instead.
- `self.cpp_info`, `self.env_info` and `self.user_info` scope has been reduced to only the `package_info()` method.
- `gcc` and `ConfigureEnvironment` were already removed in Conan 0.30.1.
- `werror` doesn’t exist anymore. It is now built-in behavior.
- Command `test_package` has been removed. Use `conan create` and `conan test` instead.
- `CMake` helper now (from Conan 0.29) only allows the `CMake(self)` syntax.
- `conan package_files` command was replaced in Conan 0.28 by `conan export-pkg` command.

16.1.3 Settings and profiles. GCC/Clang versioning

GCC and Clang compilers have modified their versioning approach, from GCC > 5 and Clang > 4. The minor versions are really bugfixes, and then they have binary compatibility. To adapt to this, Conan now includes the major version in the `settings.yml` default settings file:

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

Most package creators want to use the major-only settings, such as `-s compiler=gcc -s compiler.version=5`, instead of also specifying the minor versions.

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

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

16.1.4 New features

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

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

- Model and utilities for Windows subsystems
os:
    Windows:
        subsystem: [None, cygwin, msys, msys2, wsl]

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

16.2 General

16.2.1 Is Conan CMake based, or is CMake a requirement?

No. It isn’t. Conan is build-system agnostic. Package creators could very well use cmake to create their packages, but you will only need it if you want to build packages from source, or if there are no available precompiled packages for your system/settings. We use CMake extensively in our examples and documentation, but only because it is very convenient and most C/C++ devs are familiar with it.

16.2.2 Is build-system XXXXX supported?

Yes. It is. Conan makes no assumption about the build system. It just wraps any build commands specified by the package creators. There are already some helper methods in code to ease the use of CMake, but similar functions can be very easily added for your favorite build system. Please check out the alternatives explained in generator packages

16.2.3 Is my compiler, version, architecture, or setting supported?

Yes. Conan is very general, and does not restrict any configuration at all. However, Conan comes with some compilers, versions, architectures, ..., etc. pre-configured in the ~/.conan/settings.yml file, and you can get an error if using settings not present in that file. Go to invalid settings to learn more about it, or see the section Customizing settings.

16.2.4 Does it run offline?

Yes. It runs offline very well. Package recipes and binary packages are stored in your machine, per user, and so you can start new projects that depend on the same libraries without any Internet connection at all. Packages can be fully created, tested and consumed locally, without needing to upload them anywhere.

16.2.5 Is it possible to install 2 different versions of the same library?

Yes. You can install as many different versions of the same library as you need, and easily switch among them in the same project, or have different projects use different versions simultaneously, and without having to install/uninstall or re-build any of them.

Package binaries are stored per user in (e.g.) ~/.conan/data/Boost/1.59/user/stable/package/ (sha_0, sha_1, sha_2...) with a different SHA signature for every different configuration (debug, release, 32-bit, 64-bit, compiler...). Packages are managed per user, but additionally differentiated by version and channel, and also by their configuration. So large packages, like Boost, don’t have to be compiled or downloaded for every project.
16.2.6 Can I run multiple Conan isolated instances (virtual environments) on the same machine?

Yes, Conan supports the concept of virtual environments; so it manages all the information (packages, remotes, user credentials, ... etc.) in different, isolated environments. Check virtual environments for more details.

16.2.7 Can I run the conan_server behind a firewall (on-premises)?

Yes. Conan does not require a connection to conan.io site or any other external service at all for its operation. You can install packages from the bintray conan-center repository if you want, test them, and only after approval, upload them to your on-premises server and forget about the original repository. Or you can just get the package recipes, re-build from source on your premises, and then upload the packages to your server.

16.2.8 Can I connect to Conan remote servers through a corporate proxy?

Yes, it can be configured in your ~/.conan/conan.conf configuration file or with some environment variables. Check proxy configuration for more details.

16.2.9 Can I create packages for third-party libraries?

Of course, as long as their license allows it.

16.2.10 Can I upload closed source libraries?

Yes. As long as the resulting binary artifact can be distributed freely and free of charge, at least for educational and research purposes, and as long as you comply with all licenses and IP rights of the original authors, as well as the Terms of Service. If you want to distribute your libraries only for your paying customers, please contact us.

16.2.11 Do I always need to specify how to build the package from source?

No. But it is highly recommended. If you want, you can just directly start with the binaries, build elsewhere, and upload them directly. Maybe your build() step can download pre-compiled binaries from another source and unzip them, instead of actually compiling from sources.

16.2.12 Does Conan use semantic versioning (semver) for dependencies?

It uses a convention by which package dependencies follow semver by default; thus it intelligently avoids recompilation/repackaging if you update upstream minor versions, but will correctly do so if you update major versions upstream. This behavior can be easily configured and changed in the package_id() method of your conanfile, and any versioning scheme you desire is supported.

16.3 Using Conan

16.3.1 How to package header-only libraries?

Packaging header-only libraries is similar to other packages. Be sure to start by reading and understanding the packaging getting started guide. The main difference is that a package recipe is typically much simpler. There are different
16.3.2 When to use settings or options?

While creating a package, you may want to add different configurations and variants of the package. There are two main inputs that define packages: settings and options. Read more about them in this section.

16.3.3 How to obtain the dependents of a given package?

The search model for Conan in commands such as `conan install` and `conan info` is done from the downstream or “consumer” package as the starting node of the dependency graph and upstream.

```
$ conan info Poco/1.8.1@pocoproject/stable
```

The inverse model (from upstream to downstream) is not simple to obtain for Conan packages. This is because the dependency graph is not unique, it changes for every configuration. The graph can be different for different operating systems or just by changing some package options. So you cannot query which packages are dependent on `MyLib/0.1@user/channel`, but which packages are dependent on `MyLib/0.1@user/channel:63da998e3642b50bee33` binary package. Also, the response can contain many different binary packages for the same recipe, like `MyDependent/0.1@user/channel:packageID1... ID2... MyDependent/0.1@user/channel:packageIDN`. That is the reason why `conan info` and `conan install` need a profile (default profile or one given with `--profile`) or installation files `conanbuildinfo.txt` to look for settings and options.

In order to show the inverse graph model, the bottom node is needed to build the graph upstream and an additional node too to get the inverse list. This is usually done to get the build order in case a package is updated. For example, if we want to know the build order of the Poco dependency graph in case OpenSSL is changed we could type:
If OpenSSL is changed, we would need to rebuild it (of course) and rebuild Poco.

### 16.3.4 Packages got outdated when uploading an unchanged recipe from a different machine

Usually this is caused due to different line endings in Windows and Linux/macOS. Normally this happens when Windows uploads it with CRLF while Linux/macOS do it with only LF. Conan does not change the line endings to not interfere with user. We suggest always using LF line endings. If this issue is caused by git, it could be solved with `git config --system core.autocrlf input`.

### 16.3.5 Is there any recommendation regarding which `<user>` or `<channel>` to use in a reference?

A Conan reference is defined by the following template: `<library-name>/<library-version>@<user>/` `<channel>`

The `<user>` term in a Conan reference is basically a namespace to avoid collisions of libraries with the same name and version in the local cache and in the same remote. This field is usually populated with the author’s name of the package recipe (which could be different from the author of the library itself) or with the name of the organization creating it. Here are some examples from Conan Center:

```plaintext
OpenSSL/1.1.1@conan/stable
CLI11/1.6.1@cliutils/stable
CTRE/2.1@ctre/stable
Expat/2.2.5@pix4d/stable
FakeIt/2.0.5@gasuketsu/stable
Poco/1.9.0@pocoproject/stable
c-blosc/v1.14.4@francescalte/stable
```

In the case of the `<channel>` term, normally OSS package creators use `testing` when developing a recipe (e.g. it compiles only in few configurations) and `stable` when the recipe is ready enough to be used (e.g. it is built and tested in a wide range of configurations).

From the perspective of a library developer, channels could be used to create different scopes of your library. For example, use `rc` channel for release candidates, maybe `experimental` for those kind of features, or even `qa/testing` before the library is checked by QA department or testers.

### 16.3.6 What does “outdated from recipe” mean exactly?

In some output or commands there are references to “outdated” or “outdated from recipe”. For example, there is a flag `--outdated` in `conan search` and `conan remove` to filter by outdated packages.

When packages are created, Conan stores some metadata of the package such as the settings, the final resolution of the dependencies... and it also saves the recipe hash of the recipe contents they were generated with. This way Conan is able to know the real relation between a recipe and a package.

Basically outdated packages appear when you modify a recipe and export and/or upload it, without re-building binary packages with it. This information is displayed in yellow with:
This information is important to know if the packages are up to date with the recipe or even if the packages are still “accessible” from the recipe. That means: if the recipe has completely removed an option (it could be a setting or a requirement) but there are old packages that were generated previously with that option, those packages will be impossible to install as their package ID are calculated from the recipe file (and that option does not exist anymore).

16.3.7 How to configure the remotes priority order

The lookup remote order is defined by the command `conan remote`:

```bash
$ conan remote list
conan-center: https://conan.bintray.com [Verify SSL: True]
conan-community: https://api.bintray.com/conan/conan-community/conan [Verify SSL: True]
```

As you can see, the remote `conan-center` is listed on index 0, which means it has the highest priority when searching or installing a package, followed by `conan-community`, on index 1. To update the index order, the argument `--insert` can be added to the command `conan remote update`:

```bash
$ conan remote update conan-community https://api.bintray.com/conan/conan-community/ --insert
$ conan remote list
conan-community: https://api.bintray.com/conan/conan-community/conan [Verify SSL: True]
conan-center: https://conan.bintray.com [Verify SSL: True]
```

The `--insert` argument means index 0, the highest priority, thus the `conan-community` remote will be updated as the first remote to be used.

It’s also possible to define a specific index when adding a remote to the list:

```bash
$ conan remote add bincrafters https://api.bintray.com/conan/bincrafters/public-conan --insert 1
$ conan remote list
conan-community: https://api.bintray.com/conan/conan-community/conan [Verify SSL: True]
conan-center: https://conan.bintray.com [Verify SSL: True]
```

The `bincrafters` remote needs to be added after `conan-community`, so we need to set the remote index as 1.

16.4 Troubleshooting

16.4.1 ERROR: Missing prebuilt package

When you are installing packages (with `conan install` or `conan create`) it is possible that you get an error like the following one:
WARN: Can't find a 'libzmq/4.2.0@memsharded/testing' package for the specified options and settings:
- Settings: arch=x86_64, build_type=Release, compiler=gcc, compiler.libcxx=libstdc++,
  compiler.version=4.9, os=Windows
- Options: shared=False
- Package ID: 7fe67dff831b24bc4a8b5db678a51f1be5e44e7c

ERROR: Missing prebuilt package for 'libzmq/4.2.0@memsharded/testing'
Try to build it from sources with "--build libzmq" or read "http://docs.conan.io/en/latest/faq.html"

This means that the package recipe libzmq/4.2.0@memsharded/testing exists, but for some reason there is
no precompiled package for your current settings. Maybe the package creator didn't build and shared pre-built
packages at all and only uploaded the package recipe, or maybe they are only providing packages for some platforms
or compilers. E.g. the package creator built packages from the recipe for gcc 4.8 and 4.9, but you are using gcc 5.4.

By default, conan doesn't build packages from sources. There are several possibilities:

- You can try to build the package for your settings from sources, indicating some build policy as argument, like
  --build libzmq or --build missing. If the package recipe and the source code work for your settings
  you will have your binaries built locally and ready for use.

- If building from sources fail, you might want to fork the original recipe, improve it until it supports your
  configuration, and then use it. Most likely contributing back to the original package creator is the way to go.
  But you can also upload your modified recipe and pre-built binaries under your own username too.

### 16.4.2 ERROR: Invalid setting

It might happen sometimes, when you specify a setting not present in the defaults that you receive a message like this:

```
$ conan install . -s compiler.version=4.19 ...

ERROR: Invalid setting '4.19' is not a valid 'settings.compiler.version' value.
Possible values are ['4.4', '4.5', '4.6', '4.7', '4.8', '4.9', '5.1', '5.2', '5.3',
  '5.4', '6.1', '6.2']
Read "http://docs.conan.io/en/latest/faq/troubleshooting.html#error-invalid-setting"
```

This doesn't mean that such architecture is not supported by conan, it is just that it is not present in the actual defaults
settings. You can find in your user home folder ~/.conan/settings.yml a settings file that you can modify,
edit, add any setting or any value, with any nesting if necessary. See Customizing settings.

As long as your team or users have the same settings (you can share with them the file), everything will work. The
settings.yml file is just a mechanism so users agree on a common spelling for typically settings. Also, if you think
that some settings would be useful for many other conan users, please submit it as an issue or a pull request, so it is
included in future releases.

It is possible that some build helper, like CMake will not understand the new added settings, don't use them or even
fail. Such helpers as CMake are simple utilities to translate from conan settings to the respective build system syntax
and command line arguments, so they can be extended or replaced with your own one that would handle your own
private settings.

### 16.4.3 ERROR: Setting value not defined

When you install or create a package, it is possible to see an error like this:
**ERROR:** Hello/0.1@user/testing: 'settings.arch' value not defined

This means that the recipe defined `settings = "os", "arch", ...` but a value for the `arch` setting was not provided either in a profile or in the command line. Make sure to specify a value for it in your profile, or in the command line:

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

If you are building a pure C library with gcc/clang, you might encounter an error like this:

**ERROR:** Hello/0.1@user/testing: 'settings.compiler.libcxx' value not defined

Indeed, for building a C library, it is not necessary to define a C++ standard library. And if you provide a value, you might end with multiple packages for exactly the same binary. What has to be done is to remove such subsetting in your recipe:

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

### 16.4.4 ERROR: Failed to create process

When conan is installed via pip/PyPI, and python is installed in a path with spaces (like many times in Windows “C:/Program Files...”), conan can fail to launch. This is a known python issue, and can’t be fixed from conan. The current workarounds would be:

- Install python in a path without spaces
- Use virtualenvs. Short guide:

```
$ pip install virtualenvwrapper-win # virtualenvwrapper if not Windows
$ mkvirtualenv conan
(conan) $ pip install conan
(conan) $ conan --help
```

Then, when you will be using conan, for example in a new shell, you have to activate the virtualenv:

```
$ workon conan
(conan) $ conan --help
```

Virtualenvs are very convenient, not only for this workaround, but to keep your system clean and to avoid unwanted interaction between different tools and python projects.

### 16.4.5 ERROR: Failed to remove folder (Windows)

It is possible that operating conan, some random exceptions (some with complete tracebacks) are produced, related to the impossibility to remove one folder. Two things can happen:

- The user has some file or folder open (in a file editor, in the terminal), so it cannot be removed, and the process fails. Make sure to close files, specially if you are opening or inspecting the local conan cache.
- In Windows, the Search Indexer might be opening and locking the files, producing random, difficult to reproduce and annoying errors. Please disable the Windows Search Indexer for the conan local storage folder
16.4.6 ERROR: Error while initializing Options

When installing a Conan package and the follow error occurs:

```markdown
ERROR: conanfile.py: Error while initializing options. Please define your default_options as list or multiline string
```

Probably your Conan version is outdated. The error is related to `default_options` be used as dictionary and only can be handled by Conan >= 1.8. To fix this error, update Conan to 1.8 or higher.

16.4.7 ERROR: Error while starting Conan Server with multiple workers

When running `gunicorn` to start `conan_server` in an empty environment:

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

```
**********************************************
*          ERROR: STORAGE MIGRATION NEEDED!  *
*                                              *
*                                              *
A migration of your storage is needed, please backup first the storage directory and run:
```

```bash
$ conan_server --migrate
```

Conan Server will try to create `~/conan_server/data`, `~/conan_server/server.conf` and `~/conan_server/version.txt` at first time. However, as multiple workers are running at same time, it could result in a conflict. To fix this error, you should run:

```bash
$ conan_server --migrate
```

This command must be executed before to start the workers. It will not migrate anything, but it will populate the conan_server folder. The original discussion about this error is here.
Check https://github.com/conan-io/conan for issues and more details about development, contributors, etc.

**Important:** Conan 1.16 shouldn’t break any existing 1.0 recipe or command line invocation. If it does, please submit a report on GitHub. Read more about the *Conan stability commitment.*

### 17.1 1.16.0 (4-Jun-2019)

- Feature: The `conan upload` command can receive now the full package reference to upload a binary package. The `-p` argument is now deprecated. #5224 [Docs here](#).

- Feature: Add hooks `pre_package_info` and `post_package_info` #5223 [Docs here](#).

- Feature: New build mode `--build cascade` that forces building from sources any node with dependencies also built from sources. #5218 [Docs here](#).

- Feature: Print errors and warnings to `stderr` #5206.

- Feature: New `conan new --template=mytemplate` to initialize recipes with your own templates #5189 [Docs here](#).

- Feature: Allow using wildcards to remove system requirements sentinel from cache. #5176 [Docs here](#).

- Feature: Implement `conan.conf retry` and `retry-wait` and `CONAN_RETRY` and `CONAN_RETRY_WAIT` to configure all retries for all transfers, including upload, download, and `tools.download()` #5174 [Docs here](#).

- Feature: Support yaml lists in workspace `root` field. #5156 [Docs here](#).

- Feature: Add gcc 8.3 and 9.1 new versions to default `settings.yml` #5112.

- Feature: Retry upload or download for error in response message (e.g. status is ‘500’) #4984.

- Fix: Do not retry file transfer operations for 401 and 403 auth and permissions errors. #5278.

- Fix: Copy symlinked folder when using `merge_directories` function #5237.

- Fix: Add the ability to avoid the `/verbosity` argument in CMake command line for MSBuild #5220 [Docs here](#).

- Fix: `self.copy` with `symlinks=True` does not copy symlink if the `.conan directory is a symlink #5114 #5125.

- Fix: Export detected_os from `tools.oss (5101) #5102 [Docs here](#).

- Fix: Use `revision` as the SVN’s `peg_revision` (broken for an edge case) #5029.

- Bugfix: `--update` was not updating `python_requires` using version ranges. #5265.
• Bugfix: `visual_studio` generator only adds “.lib” extension for lib names without extension, otherwise (like “.a”) respect it. #5254

• Bugfix: Fix `conan search` command showing revisions timestamps in a different time offset than UTC. #5232

• Bugfix: Meson build-helper gets correct compiler flags, AutoTools build environment adds compiler.runtime flags #5222

• Bugfix: The `cmake_multi` generator was not managing correctly the `RelWithDebInfo` and `MinSizeRel` build types. #5221

• Bugfix: Check that registry file exists before removing it #5219

• Bugfix: do not append “-T ” if generator doesn’t support it #5201

• Bugfix: `conan download` always retrieve the sources, also with `--recipe` argument, which should only skip download binaries, not the sources. #5194

• Bugfix: Using `scm` declared in a superclass failed exporting the recipe with the error `ERROR: The conanfile.py defines more than one class level 'scm' attribute`. #5185

• Bugfix: Conan command returns 6 (Invalid configuration) also when the settings are restricted in the recipe #5178

• Bugfix: Make sure that proxy “http_proxy”, “https_proxy”, “no_proxy” vars are correctly removed if custom ones are defined in the `conan.conf`. Also, avoid using `urllib.request.getproxies()`, they are broken. #5162

• Bugfix: Use `copy()` for deploy generator so that permissions of files are preserved. Required if you want to use the deploy generator to deploy executables. #5136

17.2 1.15.2 (31-May-2019)

• Bugfix: Fix bug with python-requires not being updated with `--update` if using version-ranges. #5266

• Bugfix: Fix computation of ancestors performance regression #5260

17.3 1.15.1 (16-May-2019)

• Fix: Fix regression of `conan remote update --insert` using the same URL it had before #5110

• Fix: Fix migration of `registry.json|txt` file including reference to non existing remotes. #5103

• Bugfix: Avoid crash of commands copy, imports, editable-add for packages using python_requires #5150

17.4 1.15.0 (6-May-2019)

• Feature: Updated the generated `conanfile.py` in `conan new` to the new [conan-io/hello](https://github.com/conan-io/hello) repository #5069. Docs here

• Feature: The `MSBuild` build helper allows the parameter `toolset` with `False` value to skip the toolset adjustment. #5052. Docs here

• Feature: Add GCC 9 to default settings.yml #5046. Docs here
• Feature: You can disable broken symlinks checks when packaging using $CONAN_SKIP_BROKEN_SYMLINKS_CHECK$ env var or $config.skip_broken_symlinks_check=1$ #4991. Docs here

• Feature: New deploy generator to export files from a dependency graph to an installation folder #4972. Docs here

• Feature: Create tools.Version with _limited_ capabilities #4963. Docs here

• Feature: Default filename for workspaces: conanws.yml (used in install command) #4941. Docs here

• Feature: Add install folder to command ‘conan workspace install’ #4940. Docs here

• Feature: Add compiler.cppstd setting (mark cppstd as deprecated) #4917. Docs here

• Feature: Add a -raw argument to conan inspect command to get an output only with the value of the requested attributes #4903. Docs here

• Feature: tools.get() and tools.unzip() now handle also .gz compressed files #4883. Docs here

• Feature: Add argument --force to command profile new to overwrite existing one #4880. Docs here

• Feature: Get commit message #4877. Docs here

• Fix: Remove sudo from Travis CI template #5073. Docs here

• Fix: Handle quoted path and libraries in the premake generator #5051

• Fix: A simple addition to ensure right compiler version is found on windows. #5041

• Fix: Include CMAKE_MODULE_PATH for CMake find_dependency (#4956) #5021

• Fix: Add default_package_id_mode in the default conan.conf (#4947) #5005. Docs here

• Fix: Use back slashes for visual_studio generator instead of forward slashes #5003

• Fix: Adding subparsers.required = True makes both Py2 and Py3 print an error when no arguments are entered in commands that have subarguments #4902

• Fix: Example bare package recipe excludes conanfile.py from copy #4892

• Fix: More meaningful error message when a remote communication fails to try to download a binary package. #4888

• Bugfix: conan upload --force force also the upload of package binaries, not only recipes #5088

• BugFix: MSYS 3.x detection #5078

• Bugfix: Don’t crash when an editable declare a build_folder in the layout, but not used in a workspace #5070

• Bugfix: Made compatible the cmake_find_package_multi generator with CMake < 3.9 #5042

• Bugfix: Fix broken local development flow (conan source, conan build, conan package, conan export-pkg) with recipes with python-requires #4979

• Bugfix: ‘tar_extract’ function was failing if there was a linked folder in the working dir that matches one inside the tar file. Now we use the destination_dir as base directory to check this condition. #4965

• Bugfix: Remove package folder in conan create even when using --keep-build #4918

17.5  1.14.5 (30-Apr-2019)

• Bugfix: Uncompressing a .tgz package with a broken symlink failed while touching the destination file. #5065
• Bugfix: The symlinks compressed in a *tgz* had invalid nonzero size. #5064
• Bugfix: Fixing exception of transitive build-requires mixed with normal requires #5056

17.6 1.14.4 (25-Apr-2019)

• Bugfix: Fixed error while using Visual Studio 2019 with Ninja generator. #5028
• Bugfix: Fixed error while using Visual Studio 2019 with Ninja generator. #5025
• Bugfix: Solved errors in concurrent uploads of same recipe #5014
• Bugfix: Fixed a bug that intermittently raised `ERROR: ‘NoneType’ object has no attribute ‘file_sums’` when uploading a recipe. #5012
• Bugfix: Bug in *cmake_find_package_multi* caused *CMake* to find incorrect modules in *CMake* modules paths when only *Config* files should be taken into account. #4995
• Bugfix: Fix skipping binaries because of transitive *private* requirements #4987
• Bugfix: Fix broken local development flow (conan source, conan build, conan package, conan export-pkg) with recipes with python-requires #4983

17.7 1.14.3 (11-Apr-2019)

• Bugfix: *build-requires* and *private* requirements that resolve to a dependency that is already in the graph won’t span a new node, nor will be *build-requires* or *private*. They can conflict too. #4937

17.8 1.14.2 (11-Apr-2019)

• Bugfix: Run a full metadata migration in the cache to avoid old *null* revisions in package metadata #4934

17.9 1.14.1 (1-Apr-2019)

• Fix: Print a message for unhandled Conan errors building the API and collaborators #4869
• Bugfix: Client does not require credentials for anonymous downloads from remotes. #4872
• Bugfix: Fix a migration problem of *conan config install* for Conan versions 1.9 and older #4870
• Feature: Now Conan will crush your enemies, see them driven before you, and to hear the lamentation of their women! (April’s fools)

17.10 1.14.0 (28-Mar-2019)

• Feature: support new architectures *s390* and *s390x* #4810 . Docs here
• Feature: *--build* parameter now applies fnmatching onto the whole reference, allowing to control rebuilding in a much broader way. #4787 . Docs here
• Feature: Add config variable `general.error_on_override` and environment variable `CONAN_ERROR_ON_OVERRIDE` (defaulting to `False`) to configure if an overridden requirement should raise an error when overridden from downstream consumers. #4771. Docs here

• Feature: Allow to specify `revision_mode` for each recipe, values accepted are `scm` or `hash` (default) #4767. Docs here

• Feature: Sort library list name when calling `tools.collect_libs` #4761. Docs here

• Feature: Add `cmake_find_package_multi` generator. #4714. Docs here

• Feature: Implement `--source-folder` and `--target-folder` to `conan config install` command to select subfolder to install from the source origin, and also the destination folder within the cache. #4709. Docs here

• Feature: Implement `--update` argument for `python-requires` too. #4660

• Fix: Apply environment variables from profile and from requirements to `conan export-pkg` #4852

• Fix: Do not run `export_sources` automatically for `python_requires` #4838

• Fix: Show the correct profile name when detect a new one (#4818) #4824

• Fix: Allow using `reference` object in workspaces in templates for out of source builds #4812. Docs here

• Fix: Look for `vswhere` in `PATH` when using `tools.vswhere()` #4805

• Fix: SystemPackageTools doesn’t run `sudo` when it’s not found (#4470) #4774. Docs here

• Fix: Show warning if repo is not pristine and using SCM mode to set the revisions #4764

• Fix: avoid double call to `package()` method #4748. Docs here

• Fix: The `cmake_paths` generator now declares the `CONAN_XXX_ROOT` variables in case some exported cmake module file like `XXXConfig.cmake` has been patched with the `cmake.patch_config_paths()` to replace absolute paths to the local cache. #4719. Docs here

• Fix: Do not distribute the tests in the python package nor in the installers. #4713

• Fix: add support for CMake generator platform #4708. Docs here

• Fix: Fix corrupted packages with missing `conanmanifest.txt` files #4662

• Fix: Include information about all the configurations in the JSON generator #4657. Docs here

• Bugfix: Fixed authentication management when a server returns 401 uploading a file. #4857

• Bugfix: Fixed recipe revision detection when some error output or unexpected output was printed to the stdout running the `git` command. #4854

• Bugfix: The error output was piped to stdout causing issues while running git commands, especially during the detection of the scm revision #4853

• Bugfix: `conan export-pkg` should never resolve build-requires #4851

• bugfix: The `--build` pattern was case sensitive depending on the os file system, now it is always case sensitive, following the `conan search` behavior. #4842

• Bugfix: Fix metadata not being updated for `conan export-pkg` when using `--package-folder` #4834

• Bugfix: `--build` parameter now is always case-sensitive, previously it depended to the file system type. #4787. Docs here

• Bugfix: Raise an error if source files cannot be correctly copied to build folder because of long paths in Windows. #4766
• Bugfix: Use the same interface in `conan_basic_setup()` for the `cmake_multi` generator #4721. Docs here

17.11 1.13.3 (27-Mar-2019)

• Bugfix: Revision computation failed when a git repo was present but without commits #4830

17.12 1.13.2 (21-Mar-2019)

• Bugfix: Installing a reference with “update” and “build outdated” options raised an exception. #4790
• Bugfix: Solved bug with build-requires transitive build-requires #4783
• Bugfix: Fixed workspace crash when no layout was specified #4783
• Bugfix: Do not generate multiple `add_subdirectories()` for workspaces build-requires #4783


• Bugfix: Fix computation of graph when transitive diamonds are processed. #4737

17.14 1.13.0 (07-Mar-2019)

• Feature: Added `with_login` parameter to `tools.run_in_windows_bash()` #4673. Docs here
• Feature: The `deb` and `windows` Conan installers now use Python 3. #4663
• Feature: Allow configuring in `conan.conf` a different default `package_id` mode. #4644. Docs here
• Feature: Apply Jinja2 to layout files before parsing them #4596. Docs here
• Feature: Accept a PackageReference for the command `conan get` (argument `-p` is accepted, but hidden) #4494. Docs here
• Feature: Re-implement Workspaces based on Editable packages. #4481. Docs here
• Feature: Removed old “compatibility” mode of revisions. #4462. Docs here
• Fix: When revisions enabled, add the revision to the json output of the info/install commands. #4667
• Fix: JSON output for `multi_config` now works in `install` and `create` commands #4656
• Fix: Deprecate `cppflags` in favor of `cxxflags` in class CppInfo #4611. Docs here
• Fix: Return empty list if env variable is an empty string #4594
• Fix: `conan profile list` will now recursively list profiles. #4591
• Fix: Instance of ‘TestConan’ has no ‘install_folder’ member when exporting recipe #4585
• Fix: SCM replacement with comments below it #4580
• Fix: Remove package references associated to a remote in `registry.json` when that remote is deleted #4568
• Fix: Fixed issue with Artifactory when the anonymous user is enabled, causing the uploads to fail without requesting the user and password. #4526
• Fix: Do not allow an alias to override an existing package #4495
• Fix: Do not display the warning when there are files in the package folder (#4438), #4464
• Fix: Renamed the conan link command to conan editable to put packages into editable mode. #4481 . Docs here
• Bugfix: Solve problem with loading recipe python files in Python 3.7 because of module.__file__ = None #4669
• Bugfix: Do not attempt to upload non-existing packages, due to empty short_paths folders, or to explicit upload -p=id command. #4615
• Bugfix: Fix LIB overwrite in virtualbuildenv generator #4583
• Bugfix: Avoid str(self.settings.xxx) crash when the value is None. #4571 . Docs here
• Bugfix: Build-requires expand over the closure of the package they apply to, so they can create conflicts too. Previously, those conflicts were silently skipped, and builds would use an undetermined version and configuration of dependencies. #4514
• Bugfix: meson build type actually reflects recipe shared option #4489
• Bugfix: Fixed several bugs related to revisions. #4462 . Docs here
• Bugfix: Fixed several bugs related to the package metadata.json #4462 . Docs here

17.15 1.12.3 (18-Feb-2019)

• Fix: Fix potential downgrade from future 1.13 to 1.12 #4547
• Fix: Remove output warnings in MSBuild helper. #4518
• Fix: Revert default cmake generator on Windows (#4265) #4509 . Docs here
• Bugfix: Fixed problem with conanfile.txt [imports] sections using the ‘@’ character. #4539 . Docs here
• Bugfix: Fix search packages function when remote is called all #4502

17.16 1.12.2 (8-Feb-2019)

• Bugfix: Regression in MSBuild helper, incorrectly ignoring the conan_build.props file because of using a relative path instead of absolute one. #4488

17.17 1.12.1 (5-Feb-2019)

• Bugfix: GraphInfo parsing of existing graph_info.json files raises KeyError over "root". #4458
• Bugfix: Transitive Editable packages fail to install #4448

17.18 1.12.0 (30-Jan-2019)

• Feature: Add JSON output to ‘info’ command #4359 . Docs here
• Feature: Remove system requirements conan folders (not installed binaries) from cache #4354 . Docs here
• Feature: Updated CONTRIBUTING.md with code style #4348
• Feature: Updated OS versions for apple products #4345
• Feature: add environment variable CONAN_CACHE_NO_LOCKS to simplify debugging #4309. Docs here
• Feature: The commands conan install, conan info, conan create and conan export-pkg now can receive multiple profile arguments. The applied profile will be the composition of them, prioritizing the latest applied. #4308. Docs here
• Feature: Added get_tag() methods to tools.Git() and tools.SVN() helpers. #4306. Docs here
• Feature: Package reference is now accepted as an argument in conan install --build #4305. Docs here
• Feature: define environment variables for CTest #4299. Docs here
• Feature: Added a configuration entry at the conan.conf file to be able to specify a custom CMake executable. #4298. Docs here
• Feature: Skip “README.md” and “LICENSE.txt” during the installation of a custom config via conan config install. #4259. Docs here
• Feature: allow to specify MSBuild verbosity level #4251. Docs here
• Feature: add definitions to MSBuild build helper (and tools.build_sln_command()) #4239. Docs here
• Feature: Generate deterministic short paths on Windows #4238
• Feature: The tools.environment_append() now accepts unsetting variables by means of appending such variable with a value equal to None. #4224. Docs here
• Feature: enable a new reference argument in conan install <path> <reference>, where reference can be a partial reference too (identical to what is passed to conan create or conan export. This allows defining all pkg,version,user,channel fields of the recipe for the local flow. #4197. Docs here
• Feature: Added support for new architecture ppc32 #4195. Docs here
• Feature: Added support for new architecture armv8.3 #4195. Docs here
• Feature: Added support for new architecture armv8_32 #4195. Docs here
• Feature: Add experimental support for packages in editable mode #4181. Docs here
• Fix: Conditionally expand list-like environment variables in virtualenv generator #4396
• Fix: get_cross_building_settings for MSYS #4390
• Fix: Implemented retrial of output to stdout stream when the OS (Windows) is holding it and producing IOError for output #4375
• Fix: Validate CONAN_CPU_COUNT and output user-friendly message for invalid values #4372
• Fix: Map cpp_info.cppflags to CONAN_CXXFLAGS in make generator. #4349. Docs here
• Fix: Use *_DIRS instead of *_PATHS ending for varaibles generated by the make generator: INCLUDE_DIRS, LIB_DIRS, BIN_DIRS, BUILD_DIRS and RES_DIRS #4349. Docs here
• Fix: Bumped requirement of pyOpenSSL on OSX to >=16.0.0, <19.0.0 #4333
• Fix: Fixed a bug in the migration of the server storage to the revisions layout. #4325
• Fix: ensure tools.environment_append doesn’t raise trying to unset variables #4324. Docs here
• Fix: Improve error message when a server (like a proxy), returns 200-OK for a conan api call, but with an unexpected message. #4317
• Fix: ensure is_windows, detect_windows_subsystem, uname work under MSYS/Cygwin #4313

• Fix: uname shouldn’t use -o flag, which is GNU extention #4311

• Fix: get_branch() method of tools.SVN() helper now returns only the branch name, not the tag when present. #4306 . Docs here

• Fix: Conan client now always include the X-Checksum-Sha1 header in the file uploads, not only when checking if the file is already there with a remote supporting checksum deploy (Artifactory) #4303

• Fix: SCM optimization related to scm_folder.txt is taken into account only for packages under development. #4301

• Fix: Update premake generator, rename conanbuildinfo.premake -> conanbuildinfo.premake.lua, conan_cppdefines -> conan_defines #4296 . Docs here

• Fix: Using yaml.safe_load instead of load #4285

• Fix: Fixes default CMake generator on Windows to use MinGW Makefiles. #4281 . Docs here

• Fix: Visual Studio toolset is passed from settings to the MSBuild helper #4250 . Docs here

• Fix: Handle corner cases related to SCM with local sources optimization #4249

• Fix: Allow referring to projects created by b2 generator for dependencies with absolute paths. #4211

• Fix: Credentials are removed from SCM url attribute if Conan is automatically resolving it. #4207 . Docs here

• Fix: Remove client/server versions check on every request. Return server capabilities only in ping endpoint. #4205

• Fix: Updated contributing guidelines to the new workflow #4173

• Bugfix: Fixes config install when copying hooks #4412

• BugFix: Meson generator was failing in case of package_folder == None (test_package using Meson) #4391

• BugFix: Prepend environment variables are applied twice in conanfile #4380

• Bugfix: Caching of several internal loaders broke the conan_api usage #4362

• Bugfix: Removing usage of FileNotFoundError which is Py3 only #4361

• Bugfix: Custom generator allow to use imports #4358 . Docs here

• Bugfix: conanbuildinfo.cmake won’t fail if project() LANGUAGE is None, but the user defines CONAN_DISABLE_CHECK_COMPILER. #4276

• Bugfix: Fix version ranges containing spaces and not separated by commas. #4273

• Bugfix: When running consecutively Conan python API calls to create the default profile object became modified and cached between calls. #4256

• Bugfix: Fixes a bug in the CMake build helper about how flags are appended #4227

• Bugfix: Apply the environment to the local conan package command #4204

• Bugfix: b2 generator was failing when package recipe didn’t use compiler setting #4202

17.19 1.11.2 (8-Jan-2019)

• Bugfix: The migrated data in the server from a version previous to Conan 1.10.0 was not migrated creating the needed indexes. This fixes the migration and creates the index on the fly for fixing broken migrations. Also the server doesn’t try to migrate while running but warns the user to run conan server –migrate after doing a backup
of the data, avoiding issues when running the production servers like gunicorn where the process doesn’t accept input from the user. #4229

17.20 1.11.1 (20-Dec-2018)

• BugFix: Fix conan config install requester for zip file download #4172

17.21 1.11.0 (19-Dec-2018)

• Feature: Store verify_ssl argument in conan config install #4158. Docs here
• Feature: Tox launcher to run the test suite. #4151
• Feature: Allow --graph=file.html html output using local vis.min.js and vis.min.css resources if they are found in the local cache (can be deployed via conan config install) #4133. Docs here
• Feature: Improve client DEBUG traces with better and more complete messages. #4128
• Feature: Server prints the configuration used at startup to help debugging issues. #4128
• Feature: Allow hooks to be stored in folders #4106. Docs here
• Feature: Remove files containing Macos meta-data (files beginning by ._) #4103. Docs here
• Feature: Allow arguments in git clone for conan config install #4083. Docs here
• Feature: Display the version-ranges resolutions in a cleaner way. #4065
• Feature: allow conan export . version@user/channel and conan create . version@user/channel #4062. Docs here
• Fix: cmake_find_package generator not forwarding all dependency properties #4125
• Fix: Recent updates in python break ConfigParser with % in values, like in path names containing % (jenkins) #4122
• Fix: The property file that the MSBuild() is now generated in the build_folder instead of a temporary folder to allow more reproducible builds. #4113. Docs here
• Fix: Fixed the check of the return code from Artifactory when using the checksum deploy feature. #4100
• Fix: Evaluate always SCM attribute before exporting the recipe #4088. Docs here
• Fix: Reordered Python imports #4064
• Bugfix: In ftp_download function there is extra call to ftp.login() with empty args. This causes ftp lib to login again with empty credentials and throwing exception because authentication is required by server. #4092
• Bugfix: Take into account os_build and arch_build for search queries. #4061

17.22 1.10.2 (17-Dec-2018)

• Bugfix: Fixed bad URL schema in ApiV2 that could cause URLs collisions #4138
17.23 1.10.1 (11-Dec-2018)

- Fix: Handle some corner cases of python_requires #4099
- Bugfix: Add v1_only argument in Conan server class #4096
- Bugfix: Handle invalid use of python_requires when imported like conans.python_requires #4090

17.24 1.10.0 (4-Dec-2018)

- Feature: Add include_prerelease and loose option to version range expression #3898
- Feature: Merged “revisions” feature code in develop branch, still disabled by default until it gets stabilized. #3055
- Feature: CMake global variable to disable Conan output CONAN_CMAKE_SILENT_OUTPUT #4042
- Feature: Added new make generator. #4003
- Feature: Deploy a conan snapshot package to [test.pypi.org](https://test.pypi.org/project/conan/) for every develop commit. #4000
- Fix: Using the scm feature when Conan is not able to read the gitignored files (local optimization mechanism) print a warning to improve the debug information but not crash. #4045
- Fix: The tools.get tool (download + unzip) now supports all the arguments of the download tool. e.g: verify, retry, retry_wait etc. #4041
- Fix: Improve make generator test #4018
- Fix: Add space and dot in conan new --help #3999
- Fix: Resolve aliased packages in python_requires #3957
- Bugfix: Better checks of package reference pkg/version@user/channel, avoids bugs for conanfile in 4 nested folders and conan install path/to/the/file #4044
- Bugfix: Running Windows subsystem scripts crashed when the PATH environment variable passed as a list. #4039
- Bugfix: Fix removal of conanfile.py with conan source command and the removal of source folder in the local cache when something fails #4033
- Bugfix: A conan install with a reference failed when running in the operating system root folder because python tried to create the directory even when nothing is going to be written. #4012
- Bugfix: Fix qbs generator mixing sharedlinkflags and exelinkflags #3980
- Bugfix: compiler_args generated “mytool.lib.lib” for Visual Studio libraries that were defined with the .lib extension in the self.cpp_info.libs field of package_info(). #3976

17.25 1.9.2 (20-Nov-2018)

- Bugfix: SVN API changes are relevant since version 1.9 #3954
- Bugfix: Fixed bug in vcvars_dict tool when using filter_known_paths argument. #3941
17.26 1.9.1 (08-Nov-2018)

- Fix: Fix regression introduced in 1.7, setting amd64_x86 when no arch_build is defined. #3918
- Fix: Do not look for binaries in other remotes than the recipe, if it is defined. #3890
- Bugfix: `sudo --askpass` breaks CentOS 6 package installation. The sudo version on CentOS 6 is 1.8.6. The option of `askpass` for sudo version 1.8.7 or older is `sudo -A`. #3885

17.27 1.9.0 (30-October-2018)

- Feature: Support for `srcdirs` in `package_info()`. Packages can package sources, and specify their location, which will be propagated to consumers. Includes support for CMake generator. #3857
- Feature: Added `remote_name` and `remote_url` to upload json output. #3850
- Feature: Add environment variable `CONAN_USE_ALWAYS_SHORT_PATHS` to let the consumer override short_paths behavior from recipes #3846
- Feature: Added `--json` output to `conan export_pkg` command #3809
- Feature: Add `conan remote clean` subcommand #3767
- Feature: New `premake` generator incorporated to the Conan code base from the external generator at https://github.com/memsharded/conan-premake. #3751
- Feature: New `conan remote list_pref/add_pref/remove_pref/update_pref` commands added to manage the new Registry entries for binary packages. #3726
- Feature: Add `cpp_info` data to json output of `install` and `create` commands at package level. #3717
- Feature: Now the default templates of the `conan new` command use the docker images from the `conanio` organization: https://hub.docker.com/u/conanio #3710
- Feature: Added `topics` attribute to the `ConanFile` to specify topics (a.k.a tags, a.k.a keywords) to the recipe. #3702
- Feature: Internal refactor to the remote registry to manage a json file. Also improved internal interface. #3676
- Feature: Implement reuse of sources (`exports_sources`) in recipes used as `python_requires()`. #3661
- Feature: Added support for Clang >=8 and the new versioning schema, where only the major and the patch is used. #3643
- Fix: Renamed Plugins as Hooks #3867
- Fix: Adds GCC 8.2 to default settings.yml #3865
- Fix: Hidden confusing messages `download conaninfo.txt` when requesting the server to check package manifests. #3861
- Fix: The `MSBuild()` build helper doesn’t adjust the compiler flags for the `build_type` anymore because they are adjusted by the project itself. #3860
- Fix: Add `neon` as linux distro for SystemPackageTools #3845
- Fix: remove error that was raised for custom compiler & compiler version, while checking `cppstd` setting. #3844
- Fix: do not allow wildcards in command `conan download <ref-without-wildcards>` #3843
• Fix: do not populate arch nor arch_build in autodetected profile if platform.machine returns an empty string. #3841
• Fix: The registry won’t remove a reference to a remote removed recipe or package. #3838
• Fix: Internal improvements of the ConanFile loader #3837
• Fix: environment variables are passed verbatim to generators. #3836
• Fix: Implement dirty checks in the cache build folder, so failed builds are not packaged when there is a build_id() method. #3834
• Fix: vcvars is also called in the CMake() build helper when clang compiler is used, not only with Visual Studio’compiler. #3832
• Fix: Ignore empty line when parsing output inside SVN::excluded_files function. #3830
• Fix: Bump version of tqdm requirement to >=4.28.0 #3823
• Fix: Handling corrupted lock files in cache #3816
• Fix: Implement download concurrency checks, to allow simultaneous download of the same package (as header-only) while installing different configurations that depend on that package. #3806
• Fix: vcvars is also called in the CMake() build helper when using Ninja or NMake generators. #3803
• Fix: Fixed link_flags management in MSBuild build helper #3791
• Fix: Allow providing --profile argument (and settings, options, env, too) to conan export-pkg, so it is able to correctly compute the binary package_id in case the information captured in the installed conaninfo.txt in previous conan install does not contain all information to reconstruct the graph. #3768
• Fix: Upgrade dependency of tqdm to >=4.27: solves issue with weakref assertion. #3763
• Fix: Use XML output to retrieve information from SVN command line if its client version is less than 1.8 (command --show-item is not available). #3757
• Fix: SVN v1.7 does not have -r argument in svn status, so functionality SVN::is_pristine won’t be available. #3757
• Fix: Add --askpass argument to sudo if it is not an interactive terminal #3727
• Fix: The remote used to download a binary package is now stored, so any update for the specific binary will come from the right remote. #3726
• Fix: Use XML output from SVN command line interface to compute if the repository is pristine. #3653
• Fix: Updated templates of the conan new command with the latest conan package tools changes. #3651
• Fix: Improve error messages if conanfile was not found #3554
• BugFix: Fix conflicting multiple local imports for python_requires #3876
• Bugfix: do not ask for the username if it is already given when login into a remote. #3839
• Bugfix: yum update needs user’s confirmation, which breaks system update in CentOS non-interactive terminal. #3747

17.28 1.8.4 (19-October-2018)

• Feature: Increase debugging information when an error uploading a recipe with different timestamp occurs. #3801
• Fix: Changed `tqdm` dependency to a temporarily forked removing the “man” directory write permissions issue installing the `pip` package. #3802
• Fix: Removed `ndg-httpsclient` and `pyasn` dependencies from OSX requirements file because they shouldn’t be necessary. #3802

17.29 1.8.3 (17-October-2018)

• Feature: New attributes `default_user` and `default_channel` that can be declared in a conanfile to specify the `user` and `channel` for conan local methods when neither `CONAN_USERNAME` and `CONAN_CHANNEL` environment variables exist. #3758
• Bugfix: AST parsing of `conanfile.py` with shebang and encoding header lines was failing in python 2. This fix also allows non-ascii chars in `conanfile.py` if proper encoding is declared. #3750

17.30 1.8.2 (10-October-2018)

• Fix: Fix misleading warning message in `tools.collect_libs()` #3718
• BugFix: Fixed wrong naming of `--sbindir` and `--libexecdir` in AutoTools build helper. #3715

17.31 1.8.1 (10-October-2018)

• Fix: Remove warnings related to `python_requires()`, both in linter and due to Python2. #3706
• Fix: Use `share` folder for `DATAROOTDIR` in CMake and AutoTools build helpers. #3705
• Fix: Disabled `apiv2` until the new protocol becomes stable. #3703

17.32 1.8.0 (9-October-2018)

• Feature: Allow `conan config install` to install configuration from a folder and not only from compressed files. #3680
• Feature: The environment variable `CONAN_DEFAULT_PROFILE_PATH` allows the user to define the path (existing) to the default profile that will be used by Conan. #3675
• Feature: New `conan inspect` command that provides individual attributes of a recipe, like name, version, or options. Work with `-r=remote repos too, and is able to produce `--json` output. #3634
• Feature: Validate parameter for ConanFileReference objects to avoid unnecessary checks #3623
• Feature: The environment variable `CONAN_DEFAULT_PROFILE_PATH` allows the user to define the path (absolute and existing) to the default profile that will be used by Conan. #3615
• Feature: Warning message printed if Conan cannot deduce an architecture of a GNU triplet. #3603
• Feature: The `AutotoolsBuildEnvironment` and `CMake` build helpers now adjust default for the GNU standard installation directories: `bindir`, `sbin`, `libexec`, `includedir`, `oldincludedir`, `datarootdir` #3599
• Feature: Added `use_default_install_dirs` in `AutotoolsBuildEnvironment.configure()` to opt-out from the defaulted installation dirs. #3599
• Feature: Clean repeated entries in the PATH when vcvars is run, mitigating the max size of the env var issues. #3598
• Feature: Allow vcvars to run if clang-cl compiler is detected. #3574
• Feature: Added python 2 deprecation message in the output of the conan commands. #3567
• Feature: The conan install command now prints information about the applied configuration. #3561
• Feature: New naming convention for conanfile reserved/public/private attributes. #3560
• Feature: Experimental support for Conan plugins. #3555
• Feature: Progress bars for files unzipping. #3545
• Feature: Improved graph propagation performance from $O(n^2)$ to $O(n)$. #3528
• Feature: Added ConanInvalidConfiguration as the standard way to indicate that a specific configuration is not valid for the current package. e.g library not compatible with Windows. #3517
• Feature: Added libtool() function to the tools.XCRun() tool to locate the system libtool. #3515
• Feature: The tool tools.collect_libs() now search into each folder declared in self.cpp_info.libdirs. #3503
• Feature: Added definition CMAKE_OSX_DEPLOYMENT_TARGET to the CMake build helper following the os.version setting for Macos. #3486
• Feature: The upload of files now uses the conanmanifest.txt file to know if a file has to be uploaded or not. It avoids issues associated with the metadata of the files permissions contained in the tgz files. #3480
• Feature: The default_options in a conanfile.py can be specified now as a dictionary. #3477
• Feature: The command conan config install now support relative paths. #3468
• Feature: Added a definition CONAN_IN_LOCAL_CACHE to the CMake() build helper. #3450
• Feature: Improved AptTool at SystemPackageTool adding a function add_repository to add new apt repositories. #3445
• Feature: Experimental and initial support for the REST apiv2 that will allow transfers in one step and revisions in the future. #3442
• Feature: Improve the output of a conan install command printing dependencies when a binary is not found. #3438
• Feature: New b2 generator. It replaces the old incomplete boost_build generator that is now deprecated. #3416
• Feature: New tool.replace_path_in_file to replace Windows paths in a file doing case-insensitive comparison and indistinct path separators comparison: "/" == "" #3399
• Feature: [Experimental] Add SCM support for SVN. #3192
• Fix: None option value was not being propagated upstream in the dependency graph #3684
• Fix: Apply system_requirements() always on install, in case the folder was removed. #3647
• Fix: Included bottle package in the development requirements #3646
• Fix: More complete architecture list in the detection of the gnu triplet and the detection of the build machine architecture. #3581
• Fix: Avoid downloading the manifest of the recipe twice for uploads. Making this download quiet, without output. #3552
• Fix: Fixed Git scm class avoiding to replace any character in the get_branch() function. #3496
• Fix: Removed login username syntax checks that were no longer necessary. #3464
• Fix: Removed bad duplicated messages about dependency overriding when using conan alias. #3456
• Fix: Improve **conan info** help message. #3415
• Fix: The generator files are only written in disk if the content of the generated file changes. #3412
• Fix: Improved error message when parsing a bad conanfile reference. #3410
• Fix: Paths are replaced correctly on Windows when using `CMake().patch_config_files()`. #3399
• Fix: Fixed `AptTool` at `SystemPackageTool` to improve the detection of an installed package. #3033
• BugFix: Fixes **python_requires** overwritten when using more than one of them in a recipe #3628
• BugFix: Fix output overlap of decompress progress and plugins #3622
• Bugfix: Check if the `system_requirements()` have to be executed even when the package is retrieved from the local cache. #3616
• Bugfix: All API calls are now logged into the **CONAN_TRACE_FILE** log file. #3613
• Bugfix: Renamed `os` (reserved symbol) parameter to `os_` in the `get_gnu_triplet` tool. #3603
• Bugfix: `conan get` command now works correctly with enabled short paths. #3600
• Bugfix: Fixed `scm` replacement of the variable when exporting a conanfile. #3576
• Bugfix: `apiv2` was retrying the downloads even when a 404 error was raised. #3562
• Bugfix: Fixed `export_sources` excluded patterns containing symlinks. #3537
• Bugfix: Fixed bug with transitive private dependencies. #3525
• Bugfix: `get_cased_path` crashed when the path didn’t exist. #3516
• BugFix: Fixed failures when Conan walk directories with files containing not ASCII characters in the file name. #3505
• Bugfix: The `scm` feature now looks for the repo root even when the `conanfile.py` is in a subfolder. #3479
• Bugfix: Fixed `OSInfo.bash_path()` when there is no `windows_subsystem`. #3455
• Bugfix: AutotoolsBuildEnvironment was not defaulting the output library directory causing broken consumption of packages when rebuilding from sources in different Linux distros using lib64 default. Read more [here](#3388)

**17.33 1.7.4 (18-September-2018)**

• Bugfix: Fixed a bug in **apiv2**.
• Fix: Disabled **apiv2** by default until it gets more stability.

**17.34 1.7.3 (6-September-2018)**

• Bugfix: Uncontrolled exception was raised while printing the output of an error downloading a file.
• Bugfix: Fixed `*:option` pattern for conanfile consumers.
17.35 1.7.2 (4-September-2018)

- Bugfix: Reverted default options initialization to empty string with `varname=`.
- Bugfix: Fixed `conan build` command with `--test` and `--install` arguments.

17.36 1.7.1 (31-August-2018)

- Fix: Trailing sentences in Conan help command.
- Fix: Removed hardcoded `--init.templateDir` argument in `git clone` for `conan config install`, in favor of a new `--args` parameter that allows custom arguments.
- Fix: SCM can now handle nested subfolders.
- BugFix: Fix `conan export-pkg` unnecessarily checking remotes.

17.37 1.7.0 (29-August-2018)

- Feature: Support for C++20 in CMake > 3.12.
- Feature: Included support for Python 3.7 in all platforms.
- Feature: [Experimental] New `python_requires` function that allows you to reuse Python code by “requiring” it in Conan packages, even to extend the ConanFile class. See: [Python requires: reusing python code in recipes](https://conan.io/docs/1.7.0/conanfile.html#python-requires).
- Feature: CMake method `patch_config_paths` replaces absolute paths to a Conan package’s dependencies as well as to the Conan package itself.
- Feature: MSBuild and VisualStudioBuildEnvironment build helpers adjust the `/MP` flag to build code in parallel using multiple cores.
- Feature: Added `print_errors` parameter to `tools.PkgConfig()` helper.
- Feature: Added `--query` argument to `conan upload`.
- Feature: `virtualenv/virtualbuildenv/virtualrunenv` generators now create bash scripts in Windows for use in subsystems.
- Feature: Improved resolution speed for version ranges through caching of remote requests.
- Feature: Improved the result of `tools.vcvars_dict(only_diff=True)` including a “list” return type that can be used with `tools.environment_append()`.
- Fix: `AutoToolsBuildEnvironment` build helper now keeps the `PKG_CONFIG_PATHS` variable previously set in the environment.
- Fix: The SCM feature keeps the `.git` folder during the copy of a local directory to the local cache.
- Fix: The SCM feature now correctly excludes the folders ignored by Git during the copy of a local directory to the local cache.
- Fix: Conan messages now spell “overridden” correctly.
- Fix: MSBuild build helper arguments using quotes.
- Fix: `vcvars_command` and MSBuild build helper use the `amd64|x86` parameter when Visual Studio > 12 and when cross building for x86.
• Fix: Disabled `-c init.TemplateDir` in `conan config install` from a Git repository.
• Fix: Clang compiler check in `cmake` generator.
• Fix: Detection of Zypper package tool on latest versions of openSUSE.
• Fix: Improved help output of some commands.
• BugFix: `qmake` generator hyphen.
• Bugfix: Git submodules are now initialized from repo `HEAD` checking out the referenced revision when using the `scm` attribute.
• BugFix: Declaration `default_options` without value, e.g. `default_options = "config="`. Now it will throw an exception.
• BugFix: Deactivate script in `virtualenv` generator causes `PS1` to go unset.
• BugFix: Apply general scope options to a consumer `ConanFile` first.
• BugFix: Fixed detection of a valid repository for Git in the SCM feature.

17.38 1.6.1 (27-July-2018)

• Bugfix: `conan info --build-order` was showing duplicated nodes for build-requires and private dependencies.
• Fix: Fixed failure with the `alias` packages when the name of the package (excluded the version) was different from the aliased package. Now it is limited in the `conan alias` command.

17.39 1.6.0 (19-July-2018)

• Feature: Added a new `self.run(...)`, `run_environment=True` argument, that automatically applies `PATH`, `LD_LIBRARY_PATH` and `DYLD_LIBRARY_PATH` environment variables from the dependencies, to the execution of the current command.
• Feature: Added a new `tools.run_environment()` method as a shortcut to using `tools.environment_append` and `RunEnvironment()` together.
• Feature: Added a new `self.run(..., ignore_errors=True)` argument that represses launching an exception if the commands fails, so user can capture the return code.
• Feature: Improved `tools.Git` to allow capturing the current branch and enabling the export of a package whose version is based on the branch and commit.
• Feature: The `json` generator now outputs settings and options
• Feature: `conan remote list --raw` prints remote information in a format valid for `remotes.txt`, so it can be used for `conan config install`
• Feature: Visual Studio generator creates the `conanbuildinfo.props` file using the `$\{USERPROFILE\}` macro.
• Feature: Added a `filename` parameter to `tools.get()` in case it cannot be deduced from the URL.
• Feature: Propagated `keep_permissions` and `pattern` parameters from `tools.get()` to `tools.unzip()`.
• Feature: Added XZ extensions to `unzip()`. This will only work in Python 3 with lzma support enabled, otherwise, and error is produced.
• Feature: Added FRAMEWORK_SEARCH_PATHS var to the Xcode generator to support packaging Apple Frameworks. Read more here.

• Feature: Added conan build --test and a should_configure attribute to control the test stage. Read more here.

• Feature: New tools to convert between files with LF and CRLF line endings: tools.unix2dos() and tools.dos2unix().

• Feature: Added conan config install [url] --type=git to force cloning of a Git repo for http://... git urls.

• Feature: Improved output information when a package is missing in a remote to show which package requires the missing one.

• Feature: Improved the management of an upload interruption to avoid uploads of incomplete tarballs.

• Feature: Added new LLVM toolsets to the base settings.yml (Visual Studio).

• Feature: Created a plugin for pylint with the previous Conan checks (run in the export) enabling usage of the plugin in IDEs and command line to check if recipes are correct.

• Feature: Improved the deb installer to guarantee that it runs correctly in Debian 9 and other distros.

• Fix: Fixed conan search -q and conan remove -q to not return packages that don’t have the setting specified in the query.

• Fix: Fixed SystemPackageTool when calling to update with sudo is not enabled and mode=verify.

• Fix: Removed pyinstaller shared libraries from the linker environment for any Conan subprocess.

• BugFix: The YumTool now calls yum update instead of yum check-update.

• Bugfix: Solved a bug in which using --manifest parameter with conan create caused the deletion of information in the dependency graph.

• Bugfix: Solved bug in which the build method of the Version model was not showing the version build field correctly.

• Bugfix: Fixed a Conan crash caused by a dependency tree containing transitive private nodes.

17.40 1.5.2 (5-July-2018)

• Bugfix: Fixed bug with pre-1.0 packages with sources.

• Bugfix: Fixed regression in private requirements.

17.41 1.5.1 (29-June-2018)

• Bugfix: Sources in the local cache weren’t removed when using scm pointing to the local source directory, causing changes in local sources not applied to the conan create process.

• Bugfix: Fixed bug causing duplication of build requires in the dependency graph.

17.42 1.5.0 (27-June-2018)

• Feature: conan search <pkg-ref> --r=all now is able to search for binaries too in all remotes
• Feature: Dependency graph improvements: build_requires are represented in the graph (visible in conan info, also in the HTML graph). conan install and conan info commands shows extended information of the binaries status (represented in colors in HTML graph). The dependencies declaration order in recipes is respected (as long as it doesn’t break the dependency graph order).

• Feature: improved remote management, it is possible to get binaries from different remotes.

• Feature: conan user command is now able to show authenticated users.

• Feature: Added conan user --json json output to the command.

• Feature: New pattern argument to tools.unzip() and tools.untargz functions, that allow efficient extraction of certain files only.

• Feature: Added Manjaro support for SystemPackageTools.

• Feature: Added Macos version subsetting in the default settings.yml file, to account for the “min OSX version” configuration.

• Feature: SCM helper argument to recursively clone submodules

• Feature: SCM helper management of subfolder, allows using exports and exports_sources, manage symlinks, and do not copy files that are .gitignore. Also, works better in the local development flow.

• Feature: Modifies user agent header to output the Conan client version and the Python version. Example: Conan/1.5.0 (Python 2.7.1)

• Fix: The CMake() helper now doesn’t require a compiler input to deduce the default generator.

• Fix: conan search <pattern> now works consistently in local cache and remotes.

• Fix: Proxy related environment variables are removed if conan.conf declares proxy configuration.

• Fix: Fixed the parsing of invalid JSON when Microsoft vswhere tool outputs invalid non utf-8 text.

• Fix: Applying winsdk and vcvars_ver to MSBuild and vcvars_command for VS 14 too.

• Fix: Workspaces now support build_requires.

• Fix: CMake() helper now defines by default CMAKE_EXPORT_NO_PACKAGE_REGISTRY.

• Fix: Settings constraints declared in recipes now don’t error for single strings (instead of a list with a string element).

• Fix: cmake_minimum_required() is now before project() in templates and examples.

• Fix: CONAN_SYSREQUIRES_MODE=Disabled now doesn’t try to update the system packages registry.

• Bugfix: Fixed SCM origin path of windows folder (with backslashes).

• Bugfix: Fixed SCM dictionary order when doing replacement.

• Bugfix: Fixed auto-detection of apple-clang 10.0.

• Bugfix: Fixed bug when doing a conan search without registry file (just before installation).

17.43 1.4.5 (22-June-2018)

• Bugfix: The package_id recipe method was being called twice causing issues with info objects being populated with wrong information.
17.44 1.4.4 (11-June-2018)

- Bugfix: Fix link order with private requirements.
- Bugfix: Removed duplicate -std flag in CMake < 3 or when the standard is not yet supported by CMAKE_CXX_STANDARD.
- Bugfix: Check scm attribute to avoid breaking recipes with already defined one.
- Feature: Conan workspaces.

17.45 1.4.3 (6-June-2018)

- Bugfix: Added system libraries to the cmake_find_package generator.
- Fix: Added SIGTERM signal handler to quit safely.
- Bugfix: Fixed miss-detection of gcc 1 when no gcc was on a Linux machine.

17.46 1.4.2 (4-June-2018)

- Bugfix: Fixed multi-config packages.
- Bugfix: Fixed c++20 standard management with CMake and 20 standard version.

17.47 1.4.1 (31-May-2018)

- Bugfix: Solved issue with symlinks making recipes to fail with self.copy.
- Bugfix: Fixed c++20 standard usage with modern compilers and the creation of the settings.yml containing the settings values.
- Bugfix: Fixed error with cased directory names in Windows.
- BugFix: Modified confusing warning message in the SCM tool when the remote couldn’t be detected.

17.48 1.4.0 (30-May-2018)

- Feature: Added scm conanfile attribute, to easily clone/checkout from remote repositories and to capture the remote and commit in the exported recipe when the recipe and the sources lives in the same repository. Read more in “Recipe and sources in a different repo” and “Recipe and sources in the same repo”.
- Feature: Added cmake_paths generator to create a file setting CMAKE_MODULE_PATH and CMAKE_PREFIX_PATH to the packages folders. It can be used as a CMake toolchain to perform a transparent CMake usage, without include any line of cmake code related to Conan. Read more here.
- Feature: Added cmake_find_package generator that generates one FindXXX.cmake file per each dependency both with classic CMake approach and modern using transitive CMake targets. Read more here.
- Feature: Added conan search --json json output to the command.
- Feature: CMake build helper now sets PKG_CONFIG_PATH automatically and receives new parameter pkg_config_paths to override it.
• Feature: CMake build helper doesn’t require to specify “arch” nor “compiler” anymore when the generator is “Unix Makefiles”.
• Feature: Introduced default settings for GCC 8, Clang 7.
• Feature: Introduced support for c++ language standard c++20.
• Feature: Auto-managed fPIC option in AutoTools build helper.
• Feature: tools.vcvars_command() and tools.vcvars_dict() now take vcvars_ver and winsdk_version as parameters.
• Feature: tools.vcvars_dict() gets only the env vars set by vcvars with new parameter only_diff=True.
• Feature: Generator virtualbuildenv now sets Visual Studio env vars via tool.vcvars_dict().
• Feature: New tools for Apple development including XCRun wrapper.
• Fix: Message “Package ‘1’ created” in package commands with short_paths=True now shows package ID.
• Fix: tools.vcvars_dict() failing to create dictionary due to newlines in vcvars command output.
• Bugfix: tools.which() returning directories instead of only files.
• Bugfix: Inconsistent local cache when developing a recipe with short_paths=True.
• Bugfix: Fixed reusing MSBuild() helper object for multi-configuration packages.
• Bugfix: Fixed authentication using env vars such as CONAN_PASSWORD when CONAN_NON_INTERACTIVE=True.
• Bugfix: Fixed Android api_level was not used to adjust CMAKE_SYSTEM_VERSION.
• Bugfix: Fixed MSBuild() build helper creating empty XML node for runtime when the setting was not declared.
• Bugfix: Fixed default_options not supporting = in value when specified as tuple.
• Bugfix: AutoToolsBuildEnvironment build helper’s pkg_config_paths parameter now sets paths relative to the install folder or absolute ones if provided.

17.49 1.3.3 (10-May-2018)
• Bugfix: Fixed encoding issues writing to files and calculating md5 sums.

17.50 1.3.2 (7-May-2018)
• Bugfix: Fixed broken run_in_windows_bash due to wrong argument.
• Bugfix: Fixed VisualStudioBuildEnvironment when toolset was not defined.
• Bugfix: Fixed md5 computation of conan .tgz files for recipe, exported sources and packages due to file ordering and flags.
• Bugfix: Fixed conan download -p=wrong_id command
• Fix: Added apple-clang 9.1
17.51 1.3.1 (3-May-2018)

• Bugfix: Fixed regression with AutoToolsBuildEnvironment build helper that raised exception with not supported architectures during the calculation of the GNU triplet.
• Bugfix: Fixed pkg_config generator, previously crashing when there was no library directories in the requirements.
• Bugfix: Fixed conanfile.run() with win_bash=True quoting the paths correctly.
• Bugfix: Recovered parameter "append" to the tools.save function.
• Bugfix: Added support (documented but missing) to delete options in package_id() method using del self.info.options.<option>

17.52 1.3.0 (30-April-2018)

• Feature: Added new build types to default settings.yml: RelWithDebInfo and MinSizeRel. Compiler flags will be automatically defined in build helpers that do not understand them (MSBuild, AutotoolsBuildEnvironment)
• Feature: Improved package integrity. Interrupted downloads or builds shouldn’t leave corrupted packages.
• Feature: Added conan upload --json json output to the command.
• Feature: new conan remove --locks to clear cache locks. Useful when killing conan.
• Feature: New CircleCI template scripts can be generated with the conan new command.
• Feature: The CMake() build helper manages the fPIC flag automatically based on the options fPIC and shared when present.
• Feature: Allowing requiring color output with CONAN_COLOR_DISPLAY=1 environment variable. If CONAN_COLOR_DISPLAY is not set rely on tty detection for colored output.
• Feature: New conan remote rename and conan add --force commands to handle remotes.
• Feature: Added parameter use_env to the MSBuild().build() build helper method to control the /p:UseEnv msbuild argument.
• Feature: Timeout for downloading files from remotes is now configurable (defaulted to 60 seconds)
• Feature: Improved Autotools build helper with new parameters and automatic set of --prefix to self.package_folder.
• Feature: Added new tool to compose GNU like triplets for cross-building: tools.get_gnu_triplet()
• Fix: Use International Units for download/upload transfer sizes (Mb, Kb, etc).
• Fix: Removed duplicated paths in cmake_multi generated files.
• Fix: Removed false positive linter warning for local imports.
• Fix: Improved command line help for positional arguments
• Fix: ks alias for --keep-source argument in conan create and conan export.
• Fix: removed confusing warnings when self.copy() doesn’t copy files in the package() method.
• Fix: None is now a possible value for settings with nested subsettings in settings.yml.
• Fix: if vcvars_command is called and Visual is not found, raise an error instead of warning.
• Bugfix: self.env_info.paths and self.env_info.PATHS both map now to PATHS env-var.
• Bugfix: Local flow was not correctly recovering state for option values.
• Bugfix: Windows NTFS permissions failed in case USERDOMAIN env-var was not defined.
• Bugfix: Fixed generator pkg_config when there are absolute paths (not use prefix)
• Bugfix: Fixed parsing of settings values with "=" character in conaninfo.txt files.
• Bugfix: Fixed misdetection of MSYS environments (generation of default profile)
• Bugfix: Fixed string escaping in CMake files for preprocessor definitions.
• Bugfix: upload --no-overwrite failed when the remote package didn’t exist.
• Bugfix: Don’t raise an error if detect_windows_subsystem doesn’t detect a subsystem.

17.53 1.2.3 (10-Apr-2017)

• Bugfix: Removed invalid version field from scons generator.

17.54 1.2.1 (3-Apr-2018)

• Feature: Support for apple-clang 9.1
• Bugfix: compiler_args generator manage correctly the flag for the cppstd setting.
• Bugfix: Replaced exception with a warning message (recommending the six module) when using StringIO class from the io module.

17.55 1.2.0 (28-Mar-2018)

• Feature: The command conan build has new --configure, --build, --install arguments to control the different stages of the build() method.
• Feature: The command conan export-pkg now has a --package-folder that can be used to export an exact copy of the provided folder, irrespective of the package() method. It assumes the package has been locally created with a previous conan package or with a conan build using a cmake.install() or equivalent feature.
• Feature: New json generator, generates a json file with machine readable information from dependencies.
• Feature: Improved proxies configuration with no_proxy_match configuration variable.
• Feature: New conan upload parameter --no-overwrite to forbid the overwriting of recipe/packages if they have changed.
• Feature: Exports are now copied to source_folder when doing conan source.
• Feature: tools.vcvars() context manager has no effect if platform is different from Windows.
• Feature: conan download has new optional argument --recipe to download only the recipe of a package.
• Feature: Added CONAN_NON_INTERACTIVE environment variable to disable interactive prompts.
• Feature: Improved MSbuild() build helper using vcvars() and generating property file to adjust the run-time automatically. New method get_command() with the call to msbuild tool. Deprecates tools.build_sln_command() and tools.msvc_build_command().
• Feature: Support for clang 6.0 correctly managing cppstd flags.
• Feature: Added configuration to specify a client certificate to connect to SSL server.
• Feature: Improved ycm generator to show json dependencies.
• Feature: Experimental --json parameter for conan install and conan create to generate a JSON file with install information.
• Fix: conan install --build does not absorb more than one parameter.
• Fix: Made conanfile templates generated with conan new PEP8 compliant.
• Fix: conan search output improved when there are no packages for the given reference.
• Fix: Made conan download also retrieve sources.
• Fix: Pylint now runs as an external process.
• Fix: Made self.user and self.channel available in test_package.
• Fix: Made files writable after a deploy() or imports() when CONAN_READ_ONLY_CACHE/general.read_only_cache environment/config variable is True.
• Fix: Linter showing warnings with cpp_info object in deploy() method.
• Fix: Disabled linter for Conan pyinstaller as it was not able to find the python modules.
• Fix: conan user -r=remote_name showed all users for all remotes, not the one given.
• BugFix: Python reuse code failing to import module in package_info().
• BugFix: Added escapes for backslashes in cmake generator.
• BugFix: conan config install now raises error if git clone fails.
• BugFix: Alias resolution not working in diamond shaped dependency trees.
• BugFix: Fixed builds with Cygwin/MSYS2 failing in Windows with self.short_paths=True and NTFS file systems due to ACL permissions.
• BugFix: Failed to adjust architecture when running Conan platform detection in ARM devices.
• BugFix: Output to StringIO failing in Python 2.
• BugFix: conan profile update not working to update [env] section.
• BugFix: conan search not creating default remotes when running it as the very first command after Conan installation.
• BugFix: Package folder was not cleaned after the installation and download of a package had failed.

17.56 1.1.1 (5-Mar-2018)

• Feature: build_sln_command() and msvc_build_command() receive a new optional parameter platforms to match the definition of the .sln Visual Studio project architecture. (Typically Win32 vs x86 problem).
• Bugfix: Visual Studio runtime flags adjusted correctly in AutoToolsBuildEnvironment() build helper
• Bugfix: AutoToolsBuildEnvironment() build helper now adjust the correct build flag, not using eabi suffix, for architecture x86.
17.57 1.1.0 (27-Feb-2018)

- Feature: New `conan create --keep-build` option that allows re-packaging from conan local cache, without re-building.

- Feature: `conan search <pattern> -r=all` now searches in all defined remotes.

- Feature: Added setting `cppstd` to manage the C++ standard. Also improved build helpers to adjust the standard automatically when the user activates the setting. `AutoToolsBuildEnvironment()`, `CMake()`, `MSBuild()` and `VisualStudioBuildEnvironment()`.

- Feature: New `compiler_args` generator, for directly calling the compiler from command line, for multiple compilers: VS, gcc, clang.

- Feature: Defined `sysrequires_mode` variable (`CONAN_SYSREQUIRES_MODE` env-var) with values enabled, verify, disabled to control the installation of system dependencies via `SystemPackageTool` typically used in `system_requirements()`.

- Feature: automatically apply `pythonpath` environment variable for dependencies containing python code to be reused to recipe `source()`, `build()`, `package()` methods.

- Feature: `CMake` new `patch_config_paths()` methods that will replace absolute paths to conan package path variables, so cmake find scripts are relocatable.

- Feature: new `--test-build-folder` command line argument to define the location of the `test_package` build folder, and new conan.conf `temp_test_folder` and environment variable `CONAN_TEMP_TEST_FOLDER`, that if set to True will automatically clean the test_package build folder after running.

- Feature: Conan manages relative urls for upload/download to allow access the server from different configured networks or in domain subdirectories.

- Feature: Added `CONAN_SKIP_VS_PROJECTS_UPGRADE` environment variable to skip the upgrade of Visual Studio project when using `tools.build_sln_command()` [DEPRECATED], the `msvc_build_command` and the `MSBuild()` build helper.

- Feature: Improved detection of Visual Studio installations, possible to prioritize between multiple installed Visual tools with the `CONAN_VS_INSTALLATION_PREFERENCE` env-var and `vs_installation_preference` conan.conf variable.

- Feature: Added `keep_path` parameter to `self.copy()` within the imports() method.

- Feature: Added `[build_requires]` section to `conanfile.txt`.

- Feature: Added new `conan help <command>` command, as an alternative to `--help`.

- Feature: Added `target` parameter to `AutoToolsBuildEnvironment.make` method, allowing to select build target on running make

- Feature: The `CONAN_MAKE_PROGRAM` environment variable now it is used by the `CMake()` build helper to set a custom make program.

- Feature: Added `--verify-ssl` optional parameter to `conan config install` to allow self-signed SSL certificates in download.

- Feature: `tools.get_env()` helper method to automatically convert environment variables to python types.

- Fix: Added a visible warning about `libcxx` compatibility and the detected one for the default profile.

- Fix: Wrong detection of compiler in OSX for gcc frontend to clang.

- Fix: Disabled `conanbuildinfo.cmake` compiler checks for unknown compilers.

- Fix: `visual_studio` generator added missing `ResourceCompile` information.
• Fix: Don’t output password from URL for conan config install command.
• Fix: Signals exit with error code instead of 0.
• Fix: Added package versions to generated SCons file.
• Fix: Error message when package was not found in remotes has been improved.
• Fix: conan profile help message.
• Fix: Use gcc architecture flags -m32, -m64 for MinGW as well.
• Fix: CMake() helper do not require settings if CONAN_CMAKE_GENERATOR is defined.
• Fix: improved output of package remote origins.
• Fix: Profiles files use same structure as conan profile show command.
• Fix: conanpath.bat file is removed after conan Windows installer uninstall.
• Fix: Do not add GCC-style flags -m32, -m64, -g, -s to MSVC when using AutoToolsBuildEnvironment
• Fix: “Can’t find a binary package” message now includes the Package ID.
• Fix: added clang 5.0 and gcc 7.3 to default settings.yml.
• Bugfix: build_id() logic does not apply unless the build_id is effectively changed.
• Bugfix: self.install_folder was not correctly set in all necessary cases.
• Bugfix: --update option does not ignore local packages for version-ranges.
• Bugfix: Set self.develop=True for export-pkg command.
• Bugfix: Server HTTP responses were incorrectly captured, not showing errors for some server errors.
• Bugfix: Fixed config section update for sequential calls over the python API.
• Bugfix: Fixed wrong self.develop set to False for conan create with test_package.
• Deprecation: Removed conan-transit from default remotes registry.

17.58 1.0.4 (30-January-2018)

• Bugfix: Fixed default profile defined in conan.conf that includes another profile
• Bugfix: added missing management of sysroot in conanbuildinfo.txt affecting conan build and test_package.
• Bugfix: Fixed warning in conan source because of incorrect management of settings.
• Bugfix: Fixed priority order of environment variables defined in included profiles
• Bugfix: NMake error for parallel builds from the CMake build helper have been fixed
• Bugfix: Fixed options pattern not applied to root node (-o *:shared=True not working for consuming package)
• Bugfix: Fixed shadowed options by package name (-o *:shared=True -o Pkg:other=False was not applying shared value to Pkg)
• Fix: Using filter_known_paths=False as default to vcvars_dict() helper.
• Fix: Fixed wrong package name for output messages regarding build-requires
• Fix: Added correct metadata to conan.exe when generated via pyinstaller
17.59 1.0.3 (22-January-2018)

- Bugfix: Correct load of stored settings in conaninfo.txt (for conan build) when configure() remove some setting.
- Bugfix: Correct use of unix paths in Windows subsystems (msys, cygwin) when needed.
- Fix: fixed wrong message for conan alias --help.
- Fix: Normalized all arguments to --xxx-folder in command line help.

17.60 1.0.2 (16-January-2018)

- Fix: Adding a warning message for simultaneous use of os and os_build settings.
- Fix: Do not raise error from conanbuildinfo.cmake for Intel MSVC toolsets.
- Fix: Added more architectures to default settings.yml arch_build setting.
- Fix: using --xxx-folder in command line help messages.
- Bugfix: using quotes for Windows bash path with spaces.
- Bugfix: vcvars/vcvars_dict not including windows and windows/system32 directories in the path.

17.61 1.0.1 (12-January-2018)

- Fix: conan new does not generate cross-building (like os_build) settings by default. They make only sense for dev-tools used as build_requires
- Fix: conaninfo.txt file does not dump settings with None values

17.62 1.0.0 (10-January-2018)

- Bugfix: Fixed bug from remove_from_path due to Windows path backslash
- Bugfix: Compiler detection in conanbuildinfo.cmake for Visual Studio using toolchains like LLVM (Clang)
- Bugfix: Added quotes to bash path.

17.63 1.0.0-beta5 (8-January-2018)

- Fix: Errors from remotes different to a 404 will raise an error. Disconnected remotes have to be removed from remotes or use explicit remote with -r myremote
- Fix: cross-building message when building different architecture in same OS
- Fix: conan profile show now shows profile with same syntax as profile files
- Fix: generated test code in conan new templates will not run example app if cross building.
- Fix: conan export-pkg uses the conanfile.py folder as the default --source-folder.
- Bugfix: conan download didn’t download recipe if there are no binaries. Force recipe download.
• Bugfix: Fixed blocked `self.run()` when stderr outputs large tests, due to full pipe.

17.64 1.0.0-beta4 (4-January-2018)

• Feature: `run_in_windows_bash` accepts a dict of environment variables to be prioritized inside the bash shell, mainly intended to control the priority of the tools in the path. Use with `vcvars` context manager and `vcvars_dict`, that returns the PATH environment variable only with the Visual Studio related directories.

• Fix: Adding all values to arch_target

• Fix: `conan new` templates now use new `os_build` and `arch_build` settings

• Fix: Updated CMake helper to account for `os_build` and `arch_build` new settings

• Fix: Automatic creation of `default` profile when it is needed by another one (like `include(default)`)

• BugFix: Failed installation (non existing package) was leaving lock files in the cache, reporting a package for `conan search`.

• BugFix: Environment variables are now applied to `build_requirements()` for `conan install`.

• BugFix: Dependency graph was raising conflicts for diamonds with alias packages.

• BugFix: Fixed `conan export-pkg` after a `conan install` when recipe has options.

17.65 1.0.0-beta3 (28-December-2017)

• Fix: Upgraded pylint and astroid to latest

• Fix: Fixed `build_requires` with transitive dependencies to other build_requires

• Fix: Improved pyinstaller creation of executable, to allow for py3-64 bits (windows)

• Deprecation: removed all `--some_argument`, use instead `--some-argument` in command line.

17.66 1.0.0-beta2 (23-December-2017)

• Feature: New command line UI. Most commands use now the path to the package recipe, like `conan export . user/testing` or `conan create folder/myconanfile.py user/channel`.

• Feature: Better cross-compiling. New settings model for `os_build`, `arch_build`, `os_target`, `arch_target`.

• Feature: Better Windows OSS ecosystem, with utilities and settings model for MSYS, Cygwin, Mingw, WSL

• Feature: `package()` will not warn of not copied files for known use cases.

• Feature: reduce the scope of definition of `cpp_info`, `env_info`, `user_info` attributes to `package_info()` method, to avoid unexpected errors.

• Feature: extended the use of addressing folder and conanfiles with different names for `source`, `package` and `export-pkg` commands

• Feature: added support for Zypper system package tool

• Fix: Fixed application of build requires from profiles that didn’t apply to requires in recipes

• Fix: Improved “test package” message in output log
• Fix: updated CI templates generated with conan new
• Deprecation: Removed self.copy_headers and family for the package() method
• Deprecation: Removed self.conanfile_directory attribute.

Note: This is a beta release, shouldn’t be installed unless you do it explicitly

$ pip install conan==1.0.0b2 –upgrade

Breaking changes

• The new command line UI breaks command line tools and integration. Most cases, just add a . to the command.
• Removed self.copy_headers, self.copy_libs, methods for package(). Use self.copy() instead.
• Removed self.conanfile_directory attribute. Use self.source_folder, self.build_folder, etc. instead

17.67 0.30.3 (15-December-2017)

• Reverted CMake() and Meson() build helpers to keep old behavior.
• Forced Astroid dependency to < 1.6 because of py3 issues.

17.68 0.30.2 (14-December-2017)

• Fix: CMake() and Meson() build helpers and relative directories regression.
• Fix: ycm generator, removed the access of cpp_info to generators, keeping the access to deps_cpp_info.

17.69 0.30.1 (12-December-2017)

• Feature: Introduced major versions for gcc (5, 6, 7) as defaults settings for OSS packages, as minors are compatible by default
• Feature: VisualStudioBuildEnvironment has added more compilation and link flags.
• Feature: new MSBuild() build helper that wraps the call to msvc_build_command() with the correct application of environment variables with the improved VisualStudioBuildEnvironment
• Feature: CMake and Meson build helpers got a new cache_build_dir argument for configure(cache_build_dir=None) that will be used to define a build directory while the package is being built in local cache, but not when built locally
• Feature: conanfiles got a new apply_env attribute, defaulted to True. If false, the environment variables from dependencies will not be automatically applied. Useful if you don’t want some dependency adding itself to the PATH by default, for example
• Feature: allow recipes to use and run python code installed with conan config install.
• Feature: conanbuildinfo.cmake now has KEEP_RPATHS as argument to keep the RPATHS, as opposed to old SKIP_RPATH which was confusing. Also, it uses set(CMAKE_INSTALL_NAME_DIR "") to keep the old behavior even for CMake >= 3.9
• Feature: **conan info** is able to get profile information from the previous install, instead of requiring it as input again

• Feature: **tools.unix_path** support MSYS, Cygwin, WSL path flavors

• Feature: added **destination** folder argument to **tools.get()** function

• Feature: **SystemPackageTool** for apt-get now uses **--no-install-recommends** automatically.

• Feature: **visual_studio_multi** generator now uses toolsets instead of IDE version to identify files.

• Fix: generators failures print traces to help debugging

• Fix: typos in generator names, or non-existing generator now raise an Error instead of a warning

• Fix: **short_paths** feature is active by default in Windows. If you want to opt-out, you can use

  ```
  CONAN_USER_HOME_SHORT=None
  ```

• Fix: **SystemPackageTool** doesn’t use sudo in Windows


• **Deprecation**: **conanfile_directory** shouldn’t be used anymore in recipes. Use `source_folder, build_folder, etc.

**Note**: Breaking changes

• scopes have been completely removed. You can use environment variables, or the **conanfile.develop** or **conanfile.in_local_cache** attributes instead.

• Command **test_package** has been removed. Use **conan create** instead, and **conan test** for just running package tests.

• **werror** behavior is now by default. Dependencies conflicts will now error, and have to be fixed.

• **short_paths** feature is again active by default in Windows, even with Py3.6 and system LongPathsEnabled.

• **ConfigureEnvironment** and **GCC** build helpers have been completely removed

### 17.70 0.29.2 (2-December-2017)

• Updated python cryptography requirement for OSX due the pyOpenSSL upgrade. See more: [https://pypi.org/project/pyOpenSSL/](https://pypi.org/project/pyOpenSSL/)

### 17.71 0.29.1 (23-November-2017)

• Support for OSX High Sierra

• Reverted concurrency locks to counters, removed **psutil** dependency

• Implemented migration for settings.yml (for new VS toolsets)

• Fixed encoding issues in conan_server
17.72 0.29.0 (21-November-2017)

- Feature: Support for WindowsStore (WinRT, UWP)
- Feature: Support for Visual Studio Toolsets.
- Feature: New `boost-build` generator for generic bjam (not only Boost)
- Feature: new `tools.PkgConfig` helper to parse pkg-config (.pc) files.
- Feature: Added `self.develop conanfile` variable. It is true for `conan create` packages and for local development.
- Feature: Added `self.keep_imports` to avoid removal of imported files in the `build()` method. Convenient for re-packaging.
- Feature: Autodetected MSYS2 for `SystemPackageTool`
- Feature: `AutoToolsBuildEnvironment` now auto-loads `pkg_config_path` (to use with `pkg_config generator`)
- Feature: Changed search for profiles. Profiles not found in the default profiles folder, will be searched for locally. Use `./myprofile` to force local search only.
- Feature: Parallel builds for Visual Studio (previously it was only parallel compilation within builds)
- Feature: implemented syntax to check options with `if "something" in self.options.myoption`
- Fix: Fixed CMake dependency graph when using TARGETS, that produced wrong link order for transitive dependencies.
- Fix: Trying to download the `exports_sources` is not longer done if such attribute is not defined
- Fix: Added output directories in `cmake` generator for RelWithDebInfo and MinSizeRel configs
- Fix: Locks for concurrent access to local cache now use process IDs (PIPs) to handle interruptions and inconsistent states. Also, adding messages when locking.
- Fix: Not remove the .zip file after a `conan config install` if such file is local
- Fix: Fixed CMake.test() for the Ninja generator
- Fix: Do not create local conaninfo.txt file for `conan install <pkg-ref>` commands.
- Fix: Solved issue with multiple repetitions of the same command line argument
- BugFix: Don’t rebuild conan created (with conan-create) packages when `build_policy="always"`
- BugFix: `conan copy` was always copying binaries, now can copy only recipes
- BugFix: A bug in download was causing appends instead of overwriting for repeated downloads.
- Development: Large restructuring of files (new cmd and build folders)
- Deprecation: Removed old CMake helper methods (only valid constructor is `CMake(self)`)
- Deprecation: Removed old `conan_info()` method, that was superseded by `package_id()`

Note: Breaking changes

- `CMAKE_LIBRARY_OUTPUT_DIRECTORY` definition has been introduced in `conan_basic_setup()`, it will send shared libraries .so to the `lib` folder in Linux systems. Right now it was undefined.
- Profile search logic has slightly changed. For `-pr=myprofile`, such profile will be searched both in the default folder and in the local one if not existing. Use `-pr=./myprofile` to force local search only.
• The **conan copy** command has been fixed. To copy all binaries, it is necessary to explicit **--all**, as other commands do.

• The only valid use of CMake helper is `CMake(self)` syntax.

• If using `conan_info()`, replace it with `package_id()`.

• Removed environment variable `CONAN_CMAKE_TOOLSET`, now the toolset can be specified as a subsetting of Visual Studio compiler or specified in the build helpers.

---

**17.73  0.28.1 (31-October-2017)**

• **BugFix**: Downloading `(tools.download)` of files with `content-encoding=gzip` were raising an exception because the downloaded content length didn’t match the http header `content-length`.

---

**17.74  0.28.0 (26-October-2017)**

This is a big release, with many important and core changes. Also with a huge number of community contributions, thanks very much!

• Feature: Major revamp of most conan commands, making command line arguments homogeneous. Much better development flow adapting to user layouts, with `install-folder, source-folder, build-folder, package-folder`.

• Feature: new `deploy()` method, useful for installing binaries from conan packages

• Feature: Implemented some `concurrency` support for the conan local cache. Parallel `conan install` and `conan create` for different configurations should be possible.

• Feature: options now allow patterns in command line: `-o *:myoption=myvalue` applies to all packages

• Feature: new `pc` generator that generates files from dependencies for `pkg-config`

• Feature: new `Meson` helper, similar to CMake for Meson build system. Works well with `pc` generator.

• Feature: Support for read-only cache with `CONAN_READ_ONLY_CACHE` environment variable

• Feature: new `visual_studio_multi` generator to load Debug/Release, 32/64 configs at once

• Feature: new `tools.which` helper to locate executables

• Feature: new `conan --help` layout

• Feature: allow to override compiler version in `vcvars_command`

• Feature: `conan user` interactive (and not exposed) password input for empty `-p` argument

• Feature: Support for `PacmanTool` for `system_requirements()` for ArchLinux

• Feature: Define VS toolset in CMake constructor and from environment variable `CONAN_CMAKE_TOOLSET`

• Feature: `conan create` now accepts `werror` argument

• Feature: `AutoToolsBuildEnvironment` can use `CONAN_MAKE_PROGRAM` env-var to define make program

• Feature: added xcode9 for apple-clang 9.0, clang 5 to default settings.yml

• Feature: deactivation of `short_paths` in Windows 10 with Py3.6 and long path support is automatic
• Feature: show unzip progress by percentage, not by file (do not clutters output)
• Feature: do not use sudo for system requirements if already running as root
• Feature: tools.download able to use headers/auth
• Feature: conan does not longer generate bytecode from recipes (no more .pyc, and more efficient)
• Feature: add parallel argument to build_sln_command for VS
• Feature: Show warning if vs150comntools is an invalid path
• Feature: tools.get() now has arguments for hash checking
• Fix: upload pattern now accepts Pkg/*
• Fix: improved downloader, make more robust, better streaming
• Fix: tools.patch now support adding/removal of files
• Fix: The default profile is no longer taken as a base and merged with user profile. Use explicit include(default) instead.
• Fix: properly manage x86 as cross building with autotools
• Fix: tools.unzip removed unnecessary long-paths check in Windows
• Fix: package_info() is no longer executed at install for the consumer conanfile.py
• BugFix: source folder was not being correctly removed when recipe was updated
• BugFix: fixed CMAKE_C_FLAGS_DEBUG definition in cmake generator
• BugFix: CMAKE_SYSTEM_NAME is now Darwin for iOS, watchOS and tvOS
• BugFix: xcode generator fixed handling of compiler flags
• BugFix: pyinstaller hidden import that broke .deb installer
• BugFix: conan profile list when local files matched profile names

Note: Breaking changes

This is an important release towards stabilizing conan and moving out of beta. Some breaking changes have been
done, but mostly to command line arguments, so they should be easy to fix. Package recipes or existing packages
shouldn’t break. Please update, it is very important to ease the transition of future stable releases. Do not hesitate to
ask questions, or for help if you need it. This is a possibly not complete list of things to take into account:

• The command conan install doesn’t accept cwd anymore, to change the directory where the generator
files are written, use the --install-folder parameter.
• The command conan install doesn’t accept --all anymore. Use conan download <ref> instead.
• The command conan build now requires the path to the conanfile.py (optional before)
• The command conan package not longer re-package a package in the local cache, now it only operates in
a user local folder. The recommended way to re-package a package is using conan build and then conan
export-pkg.
• Removed conan package_files in favor of a new command conan export-pkg. It requires a local
recipe with a package() method.
• The command conan source no longer operates in the local cache. now it only operates in a user local folder.
If you used conan source with a reference to workaround the concurrency, now it natively supported, you
can remove the command call and trust concurrent install processes.
• The command `conan imports` doesn’t accept `-d`, `--dest` anymore, use `--imports-folder` parameter instead.

• If you specify a profile in a conan command, like conan create or conan install the base profile `~/.conan/profiles/default` won’t be applied. Use explicit `include` to keep the old behavior.

17.75. 0.27.0 (20-September-2017)

• Feature: `conan config install <url>` new command. Will install remotes, profiles, settings, conan.conf and other files into the local conan installation. Perfect to synchronize configuration among teams

• Feature: improved traceback printing when errors are raised for more context. Configurable via `env`

• Feature: filtering out non existing directories in `cpp_info` (include, lib, etc), so some build systems don’t complain about them.

• Feature: Added include directories to ResourceCompiler and to MIDL compiler in `visual_studio` generator

• Feature: new `visual_studio_legacy` generator for Visual Studio 2008

• Feature: show path where manifests are locally stored

• Feature: `replace_in_file` now raises error if replacement is not done (opt-out parameter)

• Feature: enabled in `conan.conf [proxies]` section `no_proxy=url1,url2` configuration (to skip proxying for those URLs), as well as `http=None` and `https=None` to explicitly disable them.

• Feature: new conanfile `self.in_local_cache` attribute for conditional logic to apply in user folders local commands

• Feature: `CONAN_USER_HOME_SHORT=None` can disable the usage of `short_paths` in Windows, for modern Windows that enable long paths at the system level

• Feature: if "arm" in `self.settings.arch` is now a valid check (without casting to `str(self.settings.arch)`) 

• Feature: added `cwd`` argument to `conan source` local method.

• Fix: unzip crashed for 0 Bytes zip files

• Fix: `collect_libs` moved to the `tools` module

• Bugfix: fixed wrong regex in `deps_cpp_info` causing issues with dots and dashes in package names

• Development: Several internal refactorings (tools module, installer), testing (using VS2015 as default, removing VS 12 in testing). Conditional CI in travis for faster builds in developers, downgrading to CMake 3.7 in appveyor

• Deprecation: `dev_requires` have been removed (it was not documented, but accessible via the `requires(dev=True)` parameter. Superseded by `build_requires`.

• Deprecation: sources tgz files for exported sources no longer contain “.c_src” subfolder. Packages created with 0.27 will be incompatible with conan < 0.25

17.76. 0.26.1 (05-September-2017)

• Feature: added apple-clang 9.0 to default settings.

• Fix: `conan copy` command now supports symlinks.

• Fix: fixed removal of “export_source” folder when files have no permissions
• Bugfix: fixed parsing of conanbuildinfo.txt with package names containing dots.

17.77  0.26.0 (31-August-2017)

• Feature: conan profile command has implemented update, new, remove subcommands, with detect ‘’,
to allow creation, edition and management of profiles.

• Feature: conan package_files command now can call recipe package() method if build_folder‘’ or
source_folder‘’ arguments are defined

• Feature: graph loading algorithm improved to avoid repeating nodes. Results in much faster times for dense
graphs, and avoids duplications of private requirements.

• Feature: authentication based on environment variables. Allows very long processes without tokens being
expired.

• Feature: Definition of Visual Studio runtime setting MD or MDd is now automatic based on build type, not
necessary to default in profile.

• Feature: Capturing SystemExit to return user error codes to the system with sys.exit(code)

• Feature: Added SKIP_RPATH argument to cmake conan_basic_setup() function

• Feature: Optimized uploads, now uploads will be skipped if there are no changes, irrespective of timestamp

• Feature: Automatic detection of VS 15-2017, via both a vs15comntools variable, and using vswhere. exe

• Feature: Added NO_OUTPUT_DIRS argument to cmake conan_basic_setup() function

• Feature: Add support for Chocolatey system package manager for Windows.

• Feature: Improved in conan user home and path storage configuration, better error checks.

• Feature: export command is now able to export recipes without name or version, specifying the full reference.

• Feature: Added new default settings, Arduino, gcc-7.2

• Feature: Add conan settings to cmake generated file

• Feature: new tools.replace_prefix_in_pc_file() function to help with .pc files.

• Feature: Adding support for system package tool pkgutil on Solaris

• Feature: conan remote update now allows --insert argument to change remote order

• Feature: Add verbose definition to CMake helper.

• Fix: conan package working locally failed if not specified build_folder

• Fix: Search when using wildcards for version like Pkg/*/user/channel

• Fix: Change current working directory to the conanfile.py one before loading it, so relative python imports or
code work.

• Fix: package_files command now works with short_paths too.

• Fix: adding missing require of tested package in test_package/conanfile build() method

• Fix: path joining in vcvars_command for custom VS paths defined via env-vars

• Fix: better managing string escaping in CMake variables

• Fix: ExecutablePath assignment has been removed from the visual_studio generator.
• Fix: removing export_source folder containing exported code, fix issues with read-only files and keeps cache consistency better.
• Fix: Accept 100 return code from yum check-update
• Fix: importing *.so files from the conan new generated test templates
• Fix: progress bars display when download/uploads are not multipart (reported size 0)
• Bugfix: fixed wrong OSX DYLD_LIBRARY_PATH variable for virtual environments
• Bugfix: FileCopier had a bug that affected self.copy() commands, changing base reference directory.

17.78 0.25.1 (20-July-2017)

• Bugfix: Build requires are now applied correctly to test_package projects.
• Fix: Fixed search command to print an error when -table parameter is used without a reference.
• Fix: install() method of the CMake() helper, allows parallel building, change build folder and custom parameters.
• Fix: Controlled errors in migration, print warning if conan is not able to remove a package directory.

17.79 0.25.0 (19-July-2017)

Note: This release introduces a new layout for the local cache, with dedicated export_source folder to store the source code exported with exports_sources feature, which is much cleaner than the old .c_src subfolder. A migration is included to remove from the local cache packages with the old layout.

• Feature: new conan create command that supersedes test_package for creating and testing package. It works even without the test_package folder, and have improved management for user, channel. The test_package recipe no longer defines requires
• Feature: new conan get command that display (with syntax highlight) package recipes, and any other file from conan: recipes, conaninfo.txt, manifests, etc.
• Feature: new conan alias command that creates a special package recipe, that works like an alias or a proxy to other package, allowing easy definition and transparent management of “using the latest minor” and similar policies. Those special alias packages do not appear in the dependency graph.
• Feature: new conan search --table=file.html command that will output an html file with a graphical representation of available binaries
• Feature: created default profile, that replace the [settings_default] in conan.conf and augments it, allowing to define more things like env-vars, options, build_requires, etc.
• Feature: new self.user_info member that can be used in package_info() to define custom user variables, that will be translated to general purpose variables by generators.
• Feature: conan remove learned the --outdated argument, to remove those binary packages that are outdated from the recipe, both from local cache and remotes
• Feature: conan search learned the --outdated argument, to show only those binary packages that are outdated from the recipe, both from local cache and remotes
• Feature: Automatic management CMAKE_TOOLCHAIN_FILE in CMake helper for cross-building.
• Feature: created conan_api, a python API interface to conan functionality.
• Feature: new `cmake.install()` method of CMake helper.
• Feature: `short_paths` feature now applies also to `exports_sources`
• Feature: `SystemPackageTool` now supports FreeBSD system packages
• Feature: `buildRequires` now manage options too, also default options in package recipes
• Feature: `conan build` learned new `--package_folder` argument, useful if the build system perform the packaging
• Feature: CMake helper now defines by default `CMAKE_INSTALL_PREFIX` pointing to the current package_folder, so `cmake.install()` can transparently execute the packaging.
• Feature: improved command UX with `cwd` arguments to allow define the current directory for the command
• Feature: improved VisualStudioBuildEnvironment
• Feature: transfers now show size (MB, KB) of download/uploaded files, and current status of transfer.
• Feature: `conan new` now has arguments to generate CI scripts for Gitlab CI.
• Feature: Added `MinRelSize` and `RelWithDebInfo` management in CMake helper.
• Fix: `make mkdir, rmdir, relative_dirs` available for import from `conans` module.
• Fix: improved detection of Visual Studio default under cygwin environment.
• Fix: `package_files` now allows symlinks
• Fix: Windows installer now includes `conan_build_info` tool.
• Fix: appending environment variables instead of overwriting them when they come from different origins: upstream dependencies and profiles.
• Fix: made opt-in the check of package integrity before uploads, it was taking too much time, and provide little value for most users.
• Fix: Package recipe linter removed some false positives
• Fix: default settings from conan.conf do not fail for constrained settings in recipes.
• Fix: Allowing to define package remote with `conan remote add_ref` before download/upload.
• Fix: removed duplicated `BUILD_SHARED_LIBS` in test_package
• Fix: add “rhel” to list of distros using yum.
• Bugfix: allowing relative paths in `exports` and `exports_sources` fields
• Bugfix: allow custom user generators with underscore

17.80 0.24.0 (15-June-2017)

• Feature: `conan new` new arguments to generate Travis-CI and Appveyor files for Continuous Integration
• Feature: Profile files with `include()` and variable declaration
• Feature: Added `RelWithDebInfo/MinRelSize` to cmake generators
• Feature: Improved linter, removing false positives due to dynamic conanfile attributes
• Feature: Added `tools.ftp_download()` function for FTP retrieval
• Feature: Managing symlinks between folders.
• Feature: `conan remote add` command learned new `insert` option to add remotes in specific order.
• Feature: support multi-config in the SCons generator
• Feature: support for gcc 7.1+ detection
• Feature: tools now are using global requests and output instances. Proxies will work for tools.
download()
• Feature: json” parameter added to conan info` command to create a JSON with the build_order.
• Fix: update default repos, now pointing to Bintray.
• Fix: printing outdated from recipe also for remotes
• Fix: Fix required slash in configure_dir of AutoToolsBuildEnvironment
• Fix: command new with very short names, now errors earlier.
• Fix: better error detection for incorrect Conanfile.py letter case.
• Fix: Improved some cmake robustness using quotes to avoid cmake errors
• BugFix: Fixed incorrect firing of building due to build’‘ patterns error
• BugFix: Fixed bug with options incorrectly applied to buildRequires and crashing
• Refactor: internal refactorings toward having a python api to conan functionality

17.81 0.23.1 (05-June-2017)

• BugFix: Fixed bug while packaging symlinked folders in build folder, and target not being packaged.
• Relaxed OSX requirement of pyopenssl to <18

17.82 0.23.0 (01-June-2017)

• Feature: new buildRequires field and build_requirements() in package recipes
• Feature: improved commands (source, build, package, package_files) and workflows for local development of
copies in user folders.
• Feature: implemented no_copy_source attribute in recipes to avoid the copy of source code from
“source” to “build folder”. Created new self.source_folder, self.build_folder, self.
package_folder for recipes to use.
• Feature: improved qmake generator with multi-config support, resource directories
• Feature: improved exception capture and formatting for all recipe user methods exceptions
• Feature: new tools.sha256() method
• Feature: folder symlinks working now for packages and upload/download
• Feature: added set_find_paths() to cmake-multi, to set CMake FindXXX.cmake paths. This will
work only for single-config build-systems.
• Feature: using environment variables for configure(), requirements() and test() methods
• Feature: added a pylintrc environment variable in conan.conf to define a PYLINTRC file with custom
style definitions (like indents).
• Feature: fixed vcvars architecture setting
• Fix: Make cacert.pem folder use CONAN_USER_HOME if existing
• Fix: fixed options=a=b option definition
• Fix: package_files command allows force'’ argument to overwrite existing instead of failing
• BugFix: Package names with underscore when parsing conanbuildinfo.txt

17.83 0.22.3 (03-May-2017)

• Fix: Fixed CMake generator (in targets mode) with linker/exe flags like –framework XXX containing spaces.

17.84 0.22.2 (20-April-2017)

• Fix: Fixed regression with usernames starting with non-alphabetical characters, introduced by 0.22.0

17.85 0.22.1 (18-April-2017)

• Fix: “-” symbol available again in usernames.
• Fix: Added future requirement to solve an error with pyinstaller generating the Windows installer.

17.86 0.22.0 (18-April-2017)

• Feature: [build_requires] can now be declared in profiles and apply them to build packages. Those requirements are only installed if the package is required to build from sources, and do not affect its package ID hash, and it is not necessary to define them in the package recipe. Ideal for testing libraries, cross compiling toolchains (like Android), development tools, etc.
• Feature: Much improved support for cross-building. Support for cross-building to Android provided, with toolchains installable via build_requires.
• Feature: New package_files command, that is able to create binary packages directly from user files, without needing to define build() or package() methods in the the recipes.
• Feature: command conan new with a new bare’’ option that will create a minimal package recipe, usable with the package_files command.
• Feature: Improved CMake helper, with test() method, automatic setting of BUILD_SHARED_LIBS, better management of variables, support for parallel compilation in MSVC (via /MP)
• Feature: new tools.msvc_build_command() helper that both sets the Visual vcvars and calls Visual to build the solution. Also vcvars_command is improved to return non-empty string even if vcvars is set, for easier concatenation.
• Feature: Added package recipe linter, warning for potential errors and also about Python 3 incompatibilities when running from Python 2. Enabled by default can be opt-out.
• Feature: Improvements in HTML output of conan info --graph.
• Feature: allow custom path to bash, as configuration and environment variable.
• Fix: Not issuing an unused variable warning in CMake for the CONAN_EXPORTED variable
• Fix: added new mips architectures and latest compiler versions to default settings.yml
• Fix: Unified username allowed patterns to those used in package references.
• Fix: hardcoded vs15 version in tools.vcvars
• BugFix: Clean crash and improved error messages when manifests mismatch exists in conan upload.

**17.87 0.21.2 (04-April-2017)**

• Bugfix: virtualenv generator quoting environment variables in Windows.

**17.88 0.21.1 (23-March-2017)**

• BugFix: Fixed missing dependencies in AutoToolsBuildEnvironment
• BugFix: Escaping single quotes in html graph of `conan info --graph=file.html`.
• BugFix: Fixed loading of auth plugins in conan_server
• BugFix: Fixed `visual_studio` generator creating XML with dots.

**17.89 0.21.0 (21-March-2017)**

• Feature: `conan info --graph` or graph=file.html` will generate a dependency graph representation in dot or html formats.
• Feature: Added better support and tests for Solaris Sparc.
• Feature: custom authenticators are now possible in `conan_server` with plugins.
• Feature: extended `conan info` command with path information and filter by packages.
• Feature: enabled conditional binary packages removal with `conan remove` with query syntax
• Feature: enabled generation and validation of manifests from `test_package`.
• Feature: allowing options definitions in profiles
• Feature: new `RunEnvironment` helper, that makes easier to run binaries from dependent packages
• Feature: new `virtualrunenv` generator that activates environment variable for execution of binaries from installed packages, without requiring imports of shared libraries.
• Feature: adding new version modes for ABI compatibility definition in `package_id()`.
• Feature: Extended `conan new` command with new option for `exports_sources` example recipe.
• Feature: CMake helper defining parallel builds for gcc-like compilers via jN`, allowing user definition with environment variable and in conan.conf.
• Feature: `conan profile` command now show profiles in alphabetical order.
• Feature: extended `visual_studio` generator with more information and binary paths for execution with DLLs paths.
• Feature: Allowing relative paths with $PROFILE_DIR place holder in profiles
• Fix: using only file checksums to decide for modified recipe in remote, for possible concurrent builds & uploads.
conan Documentation, Release 1.16.0

- Fix: Improved build modes management, with better checks and allowing multiple definitions and mixtures of conditions
- Fix: Replaced warning for non-matching OS to one message stating the cross-build
- Fix: local conan source command (working in user folder) now properly executes the equivalent of exports functionality
- Fix: Setting command line arguments to cmake command as CMake flags, while using the TARGETS approach. Otherwise, arch flags like -m32 -m64 for gcc were not applied.
- BugFix: fixed conan imports destination folder issue.
- BugFix: Allowing environment variables with spaces
- BugFix: fix for CMake with targets usage of multiple flags.

17.90 0.20.3 (06-March-2017)

- Fix: Added opt-out for CMAKE_SYSTEM_NAME automatically added when cross-building, causing users providing their own cross-build to fail
- BugFix: Corrected usage of CONAN_CFLAGS instead of CONAN_C_FLAGS in cmake targets

17.91 0.20.2 (02-March-2017)

- Fix: Regression of visual_studio``generator using `$(ExecutablePath)` instead of `$\{ExecutablePath\}`
- Fix: Regression for build=outdated --build=Path install pattern

17.92 0.20.1 (01-March-2017)

- Fix: Disabled the use of cached settings and options from installed conaninfo.txt
- Fix: Revert the use of quotes in cmake generator for flags.
- Fix: Allow comments in artifacts.properties
- Fix: Added missing commit for CMake new helpers

17.93 0.20.0 (27-February-2017)

NOTE: It is important that if you upgrade to this version, all the clients connected to the same remote, should upgrade too. Packages created with conan>=0.20.0 might not be usable with conan older conan clients.

- Feature: Largely improved management of environment variables, declaration in package_info(), definition in profiles, in command line, per package, propagation to consumers.
- Feature: New build helpers AutotoolsBuildEnvironment, VisualStudioBuildEnvironment, which deprecate ConfigureEnvironment, with much better usage of environment variables
• Feature: New virtualbuildenv generator that will generate a composable environment with build information from installed dependencies.

• Feature: New build_id() recipe method that allows to define logic to build once, and package multiple times without building. E.g.: build once both debug and release artifacts, then package separately.

• Feature: Multi-config packages. Now packages can provide multi-configuration packages, like both debug/release artifacts in the same package, with self.cpp_info.debug.libs = [...] syntax. Not restricted to debug/release, can be used for other purposes.

• Feature: new conan config command to manage, edit, display conan.conf entries

• Feature: Improvements to CMake build helper, now it has configure() and build() methods for common operations.

• Feature: Improvements to SystemPackageTool with detection of installed packages, improved implementation, installation of multi-name packages.

• Feature: Unzip with tools.unzip maintaining permissions (Linux, OSX)

• Feature: conan info command now allows profiles too

• Feature: new tools unix_path(), escape_windows_cmd(), run_in_windows_bash(), useful for autotools projects in Win/MinGW/Msys

• Feature: new context manager tools.chdir, to temporarily change directory.

• Feature: CMake using CMAKE_SYSTEM_NAME for cross-compiling.

• Feature: Artifactory build-info extraction from traces

• Feature: Attach custom headers to artifacts uploads with an artifacts.properties file.

• Feature: allow and copy symlinks while conan export

• Fix: removing quotes in some cmake variables that were generating incorrect builds

• Fix: providing better error messages for non existing binaries, with links to the docs

• Fix: improved error messages if tools.patch failed

• Fix: adding resdirs to cpp_info propagated information, and cmake variables, for directories containing resources and other data.

• Fix: printing error messages if a build”” policy doesn’t match any package

• Fix: managing VS2017 by tools. Still the manual definition of vs150comntools required.

• Bug fix: crashes when not supported characters were dumped to terminal by logger

• Bug fix: wrong executable path in Visual Studio generator

17.94 0.19.3 (27-February-2017)

• Fix: backward compatibility for new environment variables. New features to be introduced in 0.20 will produce that conaninfo.txt will not be correctly parsed, and then package would be “missing”. This will happen for packages created with 0.20, and consumed with older than 0.19.3

NOTE: It is important that you upgrade at least to this version if you are using remotes with packages that might be created with latest conan releases (like conan.io).
17.95 0.19.2 (15-February-2017)

- Bug fix: Fixed bug with remotes behind proxies
- Bug fix: Fixed bug with `exports_sources` feature and nested folders

17.96 0.19.1 (02-February-2017)

- Bug fix: Fixed issue with `conan copy` followed by `conan upload` due to the new `exports_sources` feature.

17.97 0.19.0 (31-January-2017)

- Feature: `exports_sources` allows to snapshot sources (like `exports`) but retrieve them strictly when necessary, to build from sources. This can largely improve install times for package recipes containing sources
- Feature: new configurable `tracer` able to create structured logs of conan actions: commands, API calls, etc
- Feature: new logger for `self.run` actions, able to log information from builds and other commands to files, that can afterwards be packaged together with the binaries.
- Feature: support for Solaris SunOS
- Feature: `Version` helper improved with `patch`, `pre`, `build` capabilities to handle `1.3.4-alpha2+build1` versions
- Feature: compress level of tgz is now configurable via `CONAN_COMPRESSION_LEVEL` environment variable, default 9. Reducing it can lead to faster compression times, at the expense of slightly bigger archives
- Feature: Add `powershell` support for virtualenv generator in Windows
- Feature: Improved `system_requirements()` raising errors when failing, retrying if not successful, being able to execute in user space for local recipes
- Feature: new cmake helper macro `conan_target_link_libraries()`.
- Feature: new cmake `CONAN_EXPORTED` variable, can be used in CMakeLists.txt to differentiate building in the local conan cache as package and building in user space
- Fix: improving the caching of options from `conan install` in conaninfo.txt and precedence.
- Fix: `conan definition` of cmake output dirs has been disabled for `cmake_multi` generator
- Fix: `imports()` now uses environment variables at “conan install” (but not at “conan imports” yet)
- Fix: `conan_info()` method has been renamed to `package_id()`. Backward compatibility is maintained, but it is strongly encouraged to use the new name.
- Fix: `conan_find_libraries` now use the `NO_CMAKE_FIND_ROOT_PATH` parameter for avoiding issue while cross-compiling
- Fix: disallowing duplicate URLs in remotes, better error management
- Fix: improved error message for wildcard uploads not matching any package
- Fix: remove deprecated `platform.linux_distribution()`, using new “distro” package
- Bugfix: fixed management of `VerifySSL` parameter for remotes
- Bugfix: fixed misdetection of compiler version in conanbuildinfo.cmake for apple-clang
• Bugfix: fixed trailing slash in remotes URLs producing crashes
• Refactor: A big refactor has been do to options. Nested options are no longer supported, and option. suboption will be managed as a single string option.

This has been a huge release with contributors of 11 developers. Thanks very much to all of them!

17.98 0.18.1 (11-January-2017)

• Bug Fix: Handling of transitive private dependencies in modern cmake targets
• Bug Fix: Missing quotes in CMake macro for modern cmake targets
• Bug Fix: Handling LINK_FLAGS in cmake modern targets
• Bug Fix: Environment variables no propagating to test project with test_package command

17.99 0.18.0 (3-January-2017)

• Feature: uploads and downloads with retries on failures. This helps to avoid having to fully rebuild on CI when a network transfer fails
• Feature: added SCons generator
• Feature: support for Python 3.6, with several fixes. Added Python 3.6 to CI.
• Feature: show package dates in conan info command
• Feature: new cmake_multi generator for multi-configuration IDEs like Visual Studio and Xcode
• Feature: support for Visual Studio 2017, VS-15
• Feature: FreeBSD now passes test suite
• Feature: conan upload showing error messages or URL of remote
• Feature: wildcard or pattern upload. Useful to upload multiple packages to a remote.
• Feature: allow defining settings as environment variables. Useful for use cases like dockerized builds.
• Feature: improved help’’ messages
• Feature: cmake helper tools to launch conan directly from cmake
• Added code coverage for code repository
• Fix: conan.io badges when containing dash
• Fix: manifests errors due to generated .pyc files
• Bug Fix: unicode error messages crashes
• Bug Fix: duplicated build of same binary package for private dependencies
• Bug Fix: duplicated requirement if using version-ranges and requirements() method.

17.100 0.17.2 (21-December-2016)

• Bug Fix: ConfigureEnvironment helper ignoring libcxx setting. #791
17.101 0.17.1 (15-December-2016)

- Bug Fix: conan install --all generating corrupted packages. Thanks to @yogeva
- Improved case sensitive folder management.
- Fix: appveyor links in README.

17.102 0.17.0 (13-December-2016)

- Feature: support for modern cmake with cmake INTERFACE IMPORTED targets defined per package
- Feature: support for more advanced queries in search.
- Feature: new profile list|show command, able to list or show details of profiles
- Feature: adding preliminary support for FreeBSD
- Feature: added new description field, to document package contents.
- Feature: generation of imports manifest and conan imports --undo functionality to remove imported files
- Feature: optional SSL certificate verification for remotes, to allow self signed certificates
- Feature: allowing custom paths in profiles, so profiles can be easily shared in teams, just inside the source repository or elsewhere.
- Feature: fields user and channel now available in conan recipes. That allows to declare requirements for the same user/channel as the current package.
- Feature: improved conan.io package web, adding description.
- Fix: allow to modify cmake generator in CMake helper class.
- Fix: added strip parameter to tools.patch() utility
- Fix: removed unused dependency to Boto
- Fix: wrong line endings in Windows for conan.conf
- Fix: proper automatic use of txt and env generators in test_package
- Bug fix: solved problem when uploading python packages that generated .pyc at execution
- Bug fix: crash when duplicate requires were declared in conanfile
- Bug fix: crash with existing imported files with symlinks
- Bug fix: options missing in “copy install command to clipboard” in web

17.103 0.16.1 (05-December-2016)

- Solved bug with test_package with arguments, like scopes.
17.104 0.16.0 (19-November-2016)

**Upgrade**: The build= outdated feature had a change in the hash computation, it might report outdated binaries from recipes. You can re-build the binaries or ignore it (if you haven’t changed your recipes without re-generating binaries)

- Feature: **version ranges**. Conan now supports defining requirements with version range expressions like Pkg/ >1.2, <1.9 || 1.0.1@user/channel. Check the [version ranges reference](#) for details
- Feature: decoupled imports from normal install. Now `conan install --no-imports` skips the imports section.
- Feature: new `conan imports` command that will execute the imports section without running install
- Feature: **overriding settings per package**. Now it is possible to specify individual settings for each package. This can be specified both in the command line and in [profiles](#)
- Feature: **environment variables** definition in the command line, global and per package. This allows to define specific environment variables as the compiler (CC, CXX) for a specific package. These environment variables can also be defined in [profiles](#). Check [profiles reference](#)
- Feature: Now conan files copies handle symlinks, so files are not duplicated. This will save some space and improve download speed in some large packages. To enable it, use `self.copy(..., links=True)`
- Fix: Enabling correct use of MSYS in Windows, by using the Windows C:/... path instead of the MSYS ones
- Fix: Several fixes in `conan search`, both local and in remotes
- Fix: Manifests line endings and order fix, and hash computation fixed (it had wrong ordering)
- Fix: Removed http->https redirection in conan_server that produced some issues for SSL reversed proxies
- Fix: Taking into account “ANY” definition of settings and options
- Fix: Improved some error messages and failures to encode OS errors with unicode characters
- Update: added new arch ppc64 to default settings
- Update: updated python-requests library version
- Fix: Using `generator()` instead of compiler to decide on cmake multi-configuration for Ninja+cl builds
- Improved and completed documentation

17.105 0.15.0 (08-November-2016)

**Upgrade**: If you were using the short_paths feature in Windows for packages with long paths, please reset your local cache. You could manually remove packages or just run `conan remove "*"`

- Feature: New build= outdated functionality, that allows to build the binary packages for those dependencies whose recipe has been changed, or if the binary is not existing. Each binary package stores a hash of the recipe to know if they have to be regenerated (are outdated). This information is also provided in the `conan search <ref>` command. Useful for package creators and CI.
- Feature: Extended the short_paths feature for Windows path limit to the package folder, so package with very long paths, typically in headers in nested folder hierarchies are supported.
- Feature: New `tool.build_sln_command()` helper to `build()` Microsoft Visual Studio solution (.sln) projects
- Feature: Extended the source and package command, so together with build they can be fully executed in a user folder, as a convenience for package creation and testing.

17.104. 0.16.0 (19-November-2016)
• Feature: Extending the scope of `tools.pythonpath` to work in local commands too
• Improved the parsing of profiles and better error messages
• Not adding `-s` compiler flag for clang, as it doesn’t use it.
• Automatic generation of `conanenv.txt` in local cache, warnings if using local commands and no `conanbuildinfo.txt` and no `conanenv.txt` are present to cache the information form install
• Fix: Fixed bug when using empty initial requirements (`requires = ""`)
• Fix: Added glob hidden import to pyinstaller
• Fix: Fixed minor bugs with `short_paths` as local search not listing packages
• Fix: Fixed problem with virtual envs in Windows with paths separator (using `/` instead of `)`
• Fix: Fixed parsing of `conanbuildinfo.txt`, so the root folder for each dependency is available in local commands too
• Fix: Fixed bug in `test_package` with the test project using the `requirements()` method.

17.106 0.14.1 (20-October-2016)

• Fixed bug with `short_paths` feature in windows.
• Improved error messages for non-valid profile test files.
• Remove downloaded tgz package files from remotes after decompress them.
• Fixes bug with `install --all` and short_paths

17.107 0.14.0 (20-October-2016)

• Feature: Added profiles, as user predefined settings and environment variables (as CC and CXX for compiler paths). They are stored in files in the conan cache, so they can be easily edited, added, and shared. Use them with `conan install --profile=name`
• Feature: `short_paths` feature for Windows now also handle long paths for the final package, in case that a user library has a very long final name, with nested subfolders.
• Feature: Added `tools.cpu_count()` as a helper to retrieve the number of cores, so it can be used in concurrent builds
• Feature: Detects cycles in the dependency graph, and raise error instead of exhausting recursion limits
• Feature: Conan learned the werror” option that will raise error and stop installation under some cases treated as warnings otherwise: Duplicated dependencies, or dependencies conflicts
• Feature: New `env` generator that generates a text file with the environment variables defined by dependencies, so it can be stored. Such file is parsed by `conan build` to be able to use such environment variables for `self.deps_env_info` too, in the same way it uses the `txt` generator to load variables for `self.deps_cpp_info`.
• Fix: Do not print progress bars when output is a file
• Fix: Improved the local conan search, using options too in the query `conan search -q option=value`
• Fix: Boto dependency updated to 2.43.0 (necessary for ArchLinux)
• Fix: Simplified the conan package command, removing unused and confusing options, and more informative messages about errors and utility of this command.

• Fix: More fixes and improvements on ConfigureEnvironment, mainly for Windows

• Fix: Conan now does not generate a conanbuildinfo.txt file when doing conan install <PkgRef>.

• Bug fix: Files of a package recipe are “touched” to update their timestamps to current time when retrieved, otherwise some build systems as Ninja can have problems with them.

• Bug fix: qmake generator now uses quotes to handle paths with spaces

• Bug fix: Fixed OSInfo to return the short distro name instead of the long one.

• Bug fix: fixed transitivity of private dependencies

17.108 0.13.3 (13-October-2016)

This minor solves some problems with ConfigureEnvironment, mainly for Windows, but also fixes other things:

• Fixed concatenation problems in Windows for several environment variables. Fixed problems with path with spaces

• A batch file is created in Windows to be called, as if defined structures doesn’t seem to work in the command line.

• The vcvars_command from tools now checks the Visual Studio environment variable, if it is already set, it will check it with the current project settings, throwing an error if not matching, returning an empty command if matches.

• Added a compile_flags property to ConfigureEnvironment, to be passed in the command line to the compiler, but not as environment variables

• Added defines to environment for nix systems, it was not being handled before

• Added new tests, compiling simple projects and diamond dependencies with cmake, cl (msvc), gcc (gcc in linux, mingw in win) and clang (OSX), for a better coverage of the ConfigureEnvironment functionality.

• Fixed wrong CPP_INCLUDE_PATH, it is now CPLUS_INCLUDE_PATH

17.109 0.13.0 (03-October-2016)

IMPORTANT UPGRADE ISSUE: There was a small error in the computation of binary packages IDs, that has been addressed by conan 0.13. It affects to third level (and higher) binary packages, i.e. A and B in A->B->C->D, which binaries must be regenerated for the new hashes. If you don’t plan to provide support for older conan releases (<=0.12), which would be reasonable, you should remove all binaries first (conan remove -p, works both locally and remotely), then re-build your binaries.

Features:

• Streaming from/to disk for all uploads/downloads. Previously, this was done for memory, but conan started to have issues for huge packages (>many hundreds MBs), that sometimes could be alleviated using Python 64 bits distros. This issues should be alleviated now

• New security system that allows capturing and checking the package recipes and binaries manifests into user folders (project or any other folder). That ensures that packages cannot be replaced, hacked, forged, changed or wrongly edited, either locally or in any remote server, without notice.
• Possible to handle and reuse python code in recipes. Actually, conan can be used as a package manager for python, by adding the package path to env_info.PYTHONPATH. Useful if you want to reuse common python code between different package recipes.

• Avoiding re-compress the tgz for packages after uploads if it didn’t change.

• New command conan source that executes the source() method of a given conanfile. Very useful for CI, if desired to run in parallel the construction of different binaries.

• New propagation of cpp_info, so it now allows for capturing binary packages libraries with new collect_libs() helper, and access to created binaries to compute the package_info() in general.

• Command test_package now allows the update'' option, to automatically update dependencies.

• Added new architectures for ppc64le and detection for AArch64

• New methods for defining requires effect over binary packages ID (hash) in conan_info()

• Many bugs fixes: error in tools.download with python 3, restore correct prompt in virtualenvs, bug if removing an option in config_options(), setup.py bug...

This release has contributions from @tru, @raulbocanegra, @tivek, @mathieu, and the feedback of many other conan users, thanks very much to all of them!

17.110 0.12.0 (13-September-2016)

• Major changes to search api and commands. Decoupled the search of package recipes, from the search of binary packages.

• Fixed bug that didn’t allow to export or upload packages with settings restrictions if the restrictions didn’t match the host settings

• Allowing disabling color output with CONAN_COLOR_DISPLAY=0 environment variable, or to configure color schema for light console backgrounds with CONAN_COLOR_DARK=1 environment variable

• Imports can use absolute paths, and files copied from local conan cache to those paths will not be removed when conan install. Can be used as a way to install machine-wise things (outside conan local cache)

• More robust handling of failing transfers (network disconnect), and inconsistent status after such

• Large internal refactor for storage managers. Improved implementations and decoupling between server and client

• Fixed slow conan remove for caches with many packages due to slow deletion of empty folders

• Always allowing explicit options scopes, -o Package:option=value as well as the implicit -o option=value for current Package, for consistency

• Fixed some bugs in client-server auth process.

• Allow to extract .tar files in tools.unzip()

• Some helpers for conan_info(), as self.info.requires.clear() and removal of settings and options

17.111 0.11.1 (31-August-2016)

• New error reporting for failures in conanfiles, including line number and offending line, much easier for package creators
- Removed message requesting to create an account in conan.io for other remotes
- Removed localhost:9300 remote that was added by default mostly for demo purposes. Clarified in docs.
- Fixed usernames case-sensitivity in conan_server, due to ConfigParser it was forcing lowercase
- Handling unicode characters in remote responses, fixed crash
- Added new compilers gcc 6.2, clang 8.0 to the default settings.yml
- Bumped cryptography, boto and other conan dependencies, mostly for ArchLinux compatibility and new OSX security changes

17.112.0.11.0 (3-August-2016)

- New solution for the path length limit in Windows, more robust and complete. Package conanfile.py just have to declare an attribute short_paths=True and everything will be managed. The old approach is deprecated and totally removed, so no short_paths.conf file is necessary. It should fix also the issues with uploads/retrivals.
- New virtualenv generator that generates activate and deactivate scripts that set environment variables in the current shell. It is very useful, for example to install tools (like CMake, MinGW) with conan packages, so multiple versions can be installed in the same machine, and switch between them just by activating such virtual environments. Packages for MinGW and CMake are already available as a demo
- Command conan test_package` now executes by default a conan export with smarter package reference deduction. It is introduced as opt-out behavior.
- Command :command'export' command avoids copying test_package/build temporary files in case of export=*
- Now, package_info() allows absolute paths in includedir, libdirs and bindirs, so wrapper packages can be defined that use system or manually installed libraries.
- LDFLAGS in ConfigureEnvironment management of OSX frameworks.
- Options allow the ANY value, so such option would accept any value. For example a commit of a git repository, useful to create packages that can build any specific commit of a git repo.
- Added gcc 5.4 to the default settings, as well as MinGW options (Exceptions, threads…)
- Command conan info learned a new option to output the packages from a project dependency tree that should be rebuilt in case of a modification of a certain package. It outputs a machine readable ordered list of packages to be built in that order. Useful for CI systems.
- Better management of incomplete, dirty or failed source directories (e.g. in case of a user interrupting with Ctrl+C a git clone inside the source() method.
- Added tools for easier detection of different OS versions and distributions, as well as command wrappers to install system packages (apt, yum). They use sudo via a new environment variable CONAN_SYSREQUIRES_SUDO, so using sudo is opt-in/out, for users with different sudo needs. Useful for system_requirements()
- Deprecated the config() method (still works, for backwards compatibility), but has been replaced by a config_options() to modify options based on settings, and a configure() method for most use cases. This removes a nasty behavior of having the config() method called twice with side effects.
• Now, running a `conan install MyLib/0.1@user/channel` to directly install packages without any consuming project, is also able to generate files with the `-g` option. Useful for installing tool packages (MinGW, CMake) and generate virtualenvs.

• Many small fixes and improvements: detect compiler bug in Py3, search was crashing for remotes, `conan new` failed if the package name had a dash, etc.

• Improved some internal duplications of code, refactored many tests.

This has been a big release. Practically 100% of the released features are thanks to active users feedback and contributions. Thanks very much again to all of them!

**17.113 0.10.0 (29-June-2016)**

• `conan new` command, that creates conan package conanfile.py templates, with a `test_package` package test (-t option), also for header only packages (-i option)

• Definition of _scopes_. There is a default _dev_ scope for the user project, but any other scope (test, profile...) can be defined and used in packages. They can be used to fire extra processes (as running tests), but they do not affect the package binaries, and are not included in the package IDs (hash).

• Definition of _dev_requires_. Those are requirements that are only retrieved when the package is in _dev_ scope, otherwise they are not. They do not affect the binary packages. Typical use cases would be test libraries or build scripts.

• Allow _shorter paths_ for specific packages, which can be necessary to build packages with very long path names (e.g. Qt) in Windows.

• Support for bzip2 and gzip decompression in _tools_

• Added _package_folder_ attribute to conanfile, so the `package()` method can for example call `cmake install` to create the package.

• Added `CONAN_CMAKE_GENERATOR` environment variable that allows to override the CMake default generator. That can be useful to build with Ninja instead of the default Unix Makefiles.

• Improved `ConfigureEnvironment` with include paths in CFLAGS and CPPFLAGS, and fixed bug.

• New `conan user --clean` option, to completely remove all user data for all remotes.

• Allowed to raise _Exceptions_ in `config()` method, so it is easier for package creators to raise under non-supported configurations.

• Fixed many small bugs and other small improvements

As always, thanks very much to all contributors and users providing feedback.

**17.114 0.9.2 (11-May-2016)**

• Fixed download bug that made it specially slow to download, even crash. Thanks to github @melmdk for fixing it.

• Fixed `cmake check of CLang`, it was being skipped

• Improved _performance_. Check for updates has been removed from install, made it opt-in in `conan info` command, as it was very slow, seriously affecting performance of large projects.

• Improved internal representation of graph, also improves performance for large projects.

• Fixed bug in `conan install --update`. 
17.115 0.9 (3-May-2016)

- **Python 3** “experimental” support. Now the main conan codebase is Python 2 and 3 compatible. Python 2 still the reference platform, Python 3 stable support in next releases.

- Create and share your own custom generators for any build system or tool. With “generator packages”, you can write a generator just as any other package, upload it, modify and version it, etc. Require them by reference, as any other package, and pull it into your projects dynamically.

- **Premake4** initial experimental support via a generator package.

- Very large re-write of the documentation. New “creating packages” sections with in-source and out-source explicit examples. Please read it! :)

- Improved conan test. Renamed test to test_package both for the command and the folder, but backwards compatibility remains. Custom folder name also possible. Adapted test layout might require minor changes to your package test, automatic warnings added for your convenience.

- Upgraded pyinstaller to generate binary OS installers from 2.X to 3.1

- conan search now has command line options: less verbose, verbose, extra verbose

- Added variable with full list of dependencies in conanbuildinfo.cmake

- Several minor bugfixes (check github issues)

- Improved conan user to manage user login to multiple remotes

---

17.116 0.8.4 (28-Mar-2016)

- Fixed linker problems with the new apple-clang 7.3 due to libraries with no timestamp set.

- Added apple-clang 7.3 to default settings

- Fixed default libcxx for apple-clang in auto detection of base conan.conf

---

17.117 0.8 (15-Mar-2016)

- New conan remote command to manage remotes. Redesigned remotes architecture, now allows to work with several remotes in a more consistent, powerful and “git-like” way. New remotes registry keeps track of the remote of every installed package, and this information is shown in conan info command too. Also, it keeps different user logins for different remotes, to improve support in corporate environments running in-house servers.

- New update functionality. Now it is possible to conan install --update to update packages that became obsolete because new ones were uploaded to the corresponding remote. Conan commands as install and info show information about the status of the local packages compared with the remote ones. In this way, using latest versions during development is much more natural.

- Added new compiler.libcxx setting in order to support the different c++ standard libraries. It can take libstdc++, libstdc++11 or libc++ values to take into account different standard libraries for modern gcc and clang compilers.

- Conan virtual environment: Define a custom conan directory with CONAN_USER_HOME env variable, and have a per project or per workspace storage for your dependencies. So you can isolate your dependencies and even bundle them within your project, by just setting the CONAN_USER_HOME variable to your
<project>/deps folder, for example. This also improves support for continuous integration CI systems, in which many builds from different users could be run in parallel.

- Better conanfile download method. More stable and now checks (opt-out) the ssl certificates.
- Lots of improvements: Increased library name length limit, Improved and cleaner output messages.
- Fixed several minor bugs: removing empty folders, case sensitive exports, arm settings detection.
- Introduced the concept of “package recipe” that refers to conanfile.py and exported files.
- Improved settings display in web, with new “copy install command to clipboard” to assist in installing packages discovered in web.
- The macOS installer, problematic with latest macOS releases, has been deprecated in favor of homebrew and pip install procedures.

17.118 0.7 (5-Feb-2016)

- Custom conanfile names are allowed for developing. With file’’ option you can define the file you want to use, allowing for .conaninfo.txt or having multiple conanfile_dev.py, conanfile_test.py besides the standard conanfile.py which is used for sharing the package. Inheritance is allowed, e.g. conanfile_dev.py might extend/inherit from conanfile.py.
- New conan copy command that can be used to copy/rename packages, promote them between channels, forking other users packages.
- New all’’ and package’’ options for conan install that allows to download one, several, or all package configurations for a given reference.
- Added patch() tool to easily patch sources if necessary.
- New qmake and qbs generators
- Upload of conanfile exported files is also tgz’d, allowing fast upload/downloads of full sources if desired, avoiding retrieval of sources from externals sources.
- conan info command improved showing info of current project too
- Output of run() can be redirected to buffer string for processing, or even removed.
- Added proxy configuration to conan.conf for users behinds proxies.
- Large improvements in commands output, prefixed with package reference, and much clear.
- Updated settings for more versions of gcc and new arm architectures
- Treat dependencies includes as SYSTEM in cmake, so no warnings are raised
- Deleting source folder after conan export so no manual removal is needed
- Normalizing to CRLF generated user files in Win
- Better detection and checks for compilers as VS, apple-clang
- Fixed CMAKE_SHARED_LINKER_FLAGS typo in cmake files
- Large internal refactor in generators
17.119 0.6 (11-Jan-2016)

- New cmake variables in cmake generator to make FindPackage work better thanks to the underlaying FindLibrary. Now many FindXXX.cmake work “as-is” and the package creator does not have to create a custom override, and consumers can use packages transparently with the originals FindXXX.cmake.
- New “conan info” command that shows the full dependency graph and details (license, author, url, dependants, dependancies) for each dependency.
- New environment helper with a ConfigureEnvironment class, that is able to translate conan information to autotools configure environment definition.
- Relative importing from conanfiles now is possible. So if you have common functionality between different packages, you can reuse those python files by importing them from the conanfile.py. Note that export="…" might be necessary, as packages as to be self-contained.
- Added YouCompleteMe generator for vim auto-completion of dependencies.
- New “conanfile_directory” property that points to the file in which the conanfile.py is located. This helps if using the conanfile.py “build” method to build your own project as a project, not a package, to be able to use any workflow, out-of-source builds, etc.
- Many edits and improvements in help, docs, output messages for many commands.
- All cmake syntax in modern lowercase.
- Fixed several minor bugs: gcc detection failure when gcc not installed, missing import, copying source->build failing when symlinks.

17.120 0.5 (18-Dec-2015)

- New cmake functionality allows package creators to provide cmake finders, so that package consumers can use their CMakeLists.txt with typical FindXXX.cmake files, without any change to them. CMake CONAN_CMAKE_MODULES_PATH added, so that package creators can provide any additional cmake scripts for consumers.
- Now it is possible to generate out-of-source and multiple configuration installations for the same project, so you can switch between them without having to conan install again. Check the new workflows.
- New qmake generator (thanks @dragly).
- Improved removal/deletion of folders with shutil.rmtree, so conan remove commands and other processes requiring deletion of folders do not fail due to permissions and require manual deletion. This is an improvement, especially in Win.
- Created pip package, so conan can be installed via: pip install conan.
- Released pyinstaller code for the creation of binaries from conan python source code. Distros package creators can create packages for the conan apps easily from those binaries.
- Added md5, sha1, sha256 helpers in tools, so external downloads from conanfile.py files source() can be checked.
- Added latest gcc versions to default settings.yml.
- Added CI support for conan development: travis-ci, appveyor.
- Improved human-readability for download progress, help messages.
- Minor bug fixes.