Conan, by default, will not try to detect a profile automatically, so we need to create one. To let Conan try to guess the profile, based on the current operating system and installed tools

conan profile detect --force

This will detect the operating system, build architecture and compiler settings based on the environment. It will also set the build configuration as *Release* by default. The generated profile will be stored in the Conan home folder with name *default* and will be used by Conan in all commands by default unless another profile is specified via the command line

Conan will always set the default C++ standard as the one that the detected compiler version uses by default, except for the case of macOS using apple-clang

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

1. Using a *conanfile.txt* to build your projects using Conan is enough for simple cases, but if you need more flexibility you should use a *conanfile.py* file where you can use Python code to make things such as adding requirements dynamically, changing options depending on other options or setting options for your requirements.
2. *conanfile.py*. This file is what is typically called a **“Conan recipe”**
3. It can be used for consuming packages and also to create packages
4. it will define our requirements (both libraries and build tools) and logic to modify options and set how we want to consume those packages
5. using this file to create packages, it can define (among other things) how to download the package’s source code, how to build the binaries from those sources, how to package the binaries
6. To create the Conan recipe, we declared a new class that inherits from the ConanFile class. This class has different class attributes and methods

***conanfile.py***[***¶***](https://docs.conan.io/2/tutorial/consuming_packages/the_flexibility_of_conanfile_py.html#id3)

from conan import ConanFile

class CompressorRecipe(ConanFile):

settings = "os", "compiler", "build\_type", "arch"

generators = "CMakeToolchain", "CMakeDeps"

def ***requirements***(self):

self.requires("zlib/1.3.1")

def ***build\_requirements***(self):

self.tool\_requires("cmake/3.27.9")

The **settings** class attribute defines the project-wide variables, like the compiler, its version, or the OS itself that may change when we build our project.

This is related to how Conan manages binary compatibility as these values will affect the value of the **package ID** for Conan packages.

* The **generators** class attribute specifies which Conan generators will be run when we call the **conan install** command. In this case, we added **CMakeToolchain** and **CMakeDeps** as in the *conanfile.txt*.
* In the **requirements()** method, we use the self.requires() method to declare the zlib/1.3.1 dependency.
* In the **build\_requirements()** method, we use the self.tool\_requires() method to declare the cmake/3.27.9 dependency.

**Use the layout() method**

very time we executed a *conan install* command, we had to use the *–output-folder* argument to define where we wanted to create the files that Conan generates.

Layout() is a way to decide where we want Conan to generate the files for the build system that will allow us to decide, for example, if we want different output folders depending on the type of CMake generator we are using. You can define this directly in the *conanfile.py* inside the *layout()* method and make it work for every platform without adding more changes

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

The [validate() method](https://docs.conan.io/2/reference/conanfile/methods/validate.html#reference-conanfile-methods-validate) is evaluated when Conan loads the *conanfile.py* and you can use it to perform checks of the input settings.

If, for example, your project does not support *armv8* architecture on macOS, you can raise the *ConanInvalidConfiguration* exception to make Conan return with a special error code. This will indicate that the configuration used for settings or options is not supported.

**Generate() method**

The generate() method in Conan is used to prepare the build environment by generating all the necessary files for a project's build system. It runs after the dependency graph has been computed and before the build() method is executed.

The aim of this method is to generate all the information that could be needed while running the build step

**Build() method**

If you have a recipe that implements a build() method, then it is possible to automatically call the full conan install + cmake <configure> + cmake <build> (or call the build system that the build() method uses) flow with a single command.

Though this might not be the typical developer flow, it can be a convenient shortcut in some cases.

Let’s add this build() method to our recipe:

from conan import ConanFile

from conan.tools.cmake import CMake

class CompressorRecipe(ConanFile):

settings = "os", "compiler", "build\_type", "arch"

generators = "CMakeToolchain", "CMakeDeps"

...

def *build*(self):

cmake = CMake(self)

cmake.configure()

cmake.build()

So now we can just call **conan build .**

 it manages first to install the dependencies, then it will be calling the build() method, that calls CMake configure and build step for us. Because the conan build . does internally a conan install, it can receive the same arguments (profile, settings, options, lockfile, etc.) as conan install .

**generate() vs. build()**

|  |  |
| --- | --- |
| generate() | build() |
| **Purpose:** Prepares the build environment by creating files for native build systems. | **Purpose:** Executes the actual build commands to compile the source code |
| **Execution time:** Runs before the build() method, both during conan install and conan create. | **Execution time:** Runs after the generate() method, but only when creating a package (e.g., with conan create). |
| **Generates:** Toolchain files, CMake presets, environment scripts, and dependency-finding information. | **Builds:** Compiles the source code and may run unit tests. |

**What is Cmake?**

1. CMake is an open-source, cross-platform tool that helps automate the software build process, making it easier to compile, test, and package software across different operating systems.
2. It does this by reading simple configuration files (like CMakeLists.txt) and generating native build files (like Makefiles) for your specific system and compiler.
3. This makes complex projects more manageable and allows developers to avoid writing system-specific build scripts themselves.
4. **How it works**
   1. **Platform independence:** You write a single set of build instructions in a CMakeLists.txt file, and CMake generates the correct build files for your platform (e.g., Windows, macOS, Linux).
   2. **Build system generation:** It generates build files for native build tools like Make or [Ninja](https://www.google.com/search?sca_esv=b1ea5600979d51fe&sxsrf=AE3TifOVwl7vCzqb-gIPbUpaXG39cny-QQ%3A1760587966422&q=Ninja&sa=X&sqi=2&ved=2ahUKEwiWoNai7aeQAxWUUGcHHb53OHYQxccNegQIeBAB&mstk=AUtExfCkw597bEXR9A1K6E19TTRxkBHvPfcE3YcDFFCp6gPDbRtq0NXLSdwphvn_Syils43cQcqHuiMDKbFQZ7-loe5kXPgp1vO6G1862TaUzouc3zM2vTqKl3xa531ecL30TwZS3kgpb5er9aC2DVpciIQ6iicqpMUrkX2H0GvSpAvLGfQ&csui=3). You can then use the native build tool to compile your project or use CMake itself to run the build command.
   3. **Project management:** CMake can manage complex projects with multiple files, libraries, and dependencies.
   4. **Automation:** It automates tasks like finding installed libraries, setting compiler options, and running tests via its associated tools ctest and cpack for packaging.

**cross-compile your applications using Conan: host and build contexts**

using two different profiles, one for the machine that builds the application (Ubuntu Linux) and another for the machine that runs the application (Raspberry Pi).

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

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

Is equivalent to:

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

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

|  |  |
| --- | --- |
| ***<conan home>/profiles/default*** | ***<local folder>/profiles/raspberry***[¶](https://docs.conan.io/2/tutorial/consuming_packages/cross_building_with_conan.html#id2) |
| [settings]  **os**=Linux  **arch**=x86\_64  **build\_type**=Release  **compiler**=gcc  compiler.cppstd=gnu14  compiler.libcxx=libstdc++11  compiler.version=**9** | [settings]  **os**=Linux  **arch**=armv7hf  **compiler**=gcc  **build\_type**=Release  compiler.cppstd=gnu14  compiler.libcxx=libstdc++11  compiler.version=**9**  [buildenv]  **CC**=arm-linux-gnueabihf-gcc-9  **CXX**=arm-linux-gnueabihf-g++-9  **LD**=arm-linux-gnueabihf-ld |

[buildenv] section is used to set the environment variables that are needed to build the application.

we declare the CC, CXX and LD variables pointing to the cross-build toolchain compilers and linker, respectively.

Adding this section to the profile will invoke the *VirtualBuildEnv* generator everytime we do a **conan install**.

This generator will add that environment information to the conanbuild.sh script that we will source before building with CMake so that it can use the cross-build toolchain.

Note :

We have to activate the build environment running **source Release/generators/conanbuild.sh**. That will set the environment variables needed to locate the cross-build toolchain and build the application.

**Introduction to versioning**

 using requires with fixed versions like requires = "zlib/1.2.12". But sometimes dependencies evolve, new versions are released and consumers want to update to those versions as easily as possible.

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

**Version ranges**[**¶**](https://docs.conan.io/2/tutorial/consuming_packages/intro_to_versioning.html#version-ranges)

A requires can express a dependency to a certain range of versions for a given package, with the syntax pkgname/[version-range-expression].

|  |
| --- |
| ***conanfile.py***[***¶***](https://docs.conan.io/2/tutorial/consuming_packages/intro_to_versioning.html#id1) |
| from conan import ConanFile  class CompressorRecipe(ConanFile):  settings = "os", "compiler", "build\_type", "arch"  generators = "CMakeToolchain", "CMakeDeps"  def ***requirements***(self):  self.requires("zlib/[~1.2]") |

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

**Creating packages**

1. **source() method -** to retrieve sources from external repositories and apply patches to those sources.
2. Add requirements to your Conan packages inside the requirements() method.
3. Use the generate() method to prepare the package build, and customize the toolchain.
4. Configure settings and options in the configure() and config\_options() methods and how they affect the packages’ binary compatibility.
5. Use the build() method to customize the build process and launch the tests for the library you are packaging.
6. Select which files will be included in the Conan package using the package() method.
7. Define the package information in the package\_info() method so that consumers of this package can use it.
8. Use a test\_package to test that the Conan package can be consumed correctly.

Configure settings and options in recipe

configure() : use this method to configure which options or settings of the recipe are available.

config\_options(): This method is used to **constrain** the available options in a package **before they take a value**. If a value is assigned to a setting or option that is deleted inside this method, Conan will raise an error.

this method is executed before the configure() method.

**conan export-pkg .**

 this command will create the package in the Conan cache and test it by running the *test\_package* afterwards.

**Packages in editable mode**

|  |
| --- |
| $ conan editable add say $ conan editable list  say/1.0  Path: /Users/.../examples2/tutorial/developing\_packages/editable\_packages/say/conanfile.py |

**Versioning**

set\_version() method as a mechanism to automate the definition of the package version.

Version ranges can be defined in several places:

* In conanfile.py recipes requires, tool\_requires, test\_requires, python\_requires
* In conanfile.txt files in [requires], [tool\_requires], [test\_requires] sections
* In command line arguments like --requires= and --tool\_requires.
* In profiles [tool\_requires] section