CMakeList.txt

⇒ The file CMakeLists.txt is the input to the CMake build system for building software packages.

↳ It describe how to build the code and where to install it.

* Overall Structure and Ordering

⇒ The order in the configuration DOES Count.

1. Required CMake Version (cmake_minimum_required)
2. Package Name (project())
3. Find Other CMake/Catkin packages needed for build (find_package())
4. Enable Python module support (catkin_python_setup())
5. Message/Service/Action Generators (add_message_files(), add_service_files(), add_action_files())
6. Invoke message/service/action generation (generate_messages())
7. Specify package build info export (catkin_package())

8. Libraries / Executables to build
(add_library() / add_executable()
/ target_link_libraries())

9. Tests to build (catkin_add_gtest())

10. Install rules (install())

* CMake Version

cmake_minimum_required
(VERSION 2.8.3)

* Package name

Let say name of Project is, robot_brain

⇒ Project (robot_brain)

⇒ You can reference the project name anywhere later in the CMake script by using the Variable ${PROJECT_NAME}

* Finding Dependent CMake Packages

⇒ There is always at least one dependency on catkin:

find_package (catkin REQUIRED)

⇒ You should ~~always~~ only find_package components for which you want build flags. You should not add runtime dependencies

What does find_package() Do?

⇒ If a package is found by CMake through find_package, it results in the creation of several CMake environment variables that give information about the found package.

⇒ The names always follows the convention of

<PackageName>_<Property>

⇒ <NAME>_FOUND

↳ Set to true if library is found otherwise false.

⇒ <NAME>_INCLUDE_DIRS or <NAME>_INCLUDES

↳ This includes path exported by the package.

⇒ <NAME>_LIBRARIES or <NAME>_LIBS

↳ The libraries exported by the package.

Boost

⇒ If you are using C++ and Boost, you need to invoke find_package() on Boost and specify which aspect of Boost you are using as Components.

* Catkin_package()
  ↳ Catkin-provided CMake macro.

⇒ This function must be called before declaring any targets.

⇒ The function has 5 optional argument:

INCLUDE_DIRS
LIBRARIES
CATKIN_DEPENDS
DEPENDS
CFG_EXTRAS

* Specifying Build Targets

⇒ Build targets can take many forms, but usually they represent one of two possibilities:

- Executable Target
  ↳ Programs we can run
- Library Target
  ↳ Libraries that can be used by executable targets at build and/or runtime.

Target Naming

⇒ One can have a target renamed to something else using the set_target_properties() function:

Example

```
set_target_properties (rviz_image_view
                       PROPERTIES OUTPUT_NAME
                       image_view
                       PREFIX "")
```

This will change the name of the target rviz_image_view to image_view in the build and install output

Custom output directory

⇒ While the default output directory for executables and libraries is usually set to a reasonable value it must be customized in certain case.

Example: Library containing python bindings must be placed in a different folder to be importable in Python

```
set_target_properties(Python_module_libsay
PROPERTIES LIBRARY_OUTPUT_DIRECTORY
${CATKIN_DEVEL_PREFIX}/${CATKIN_PACKAGE
_PYTHON_DESTINATION})
```

Include Path and Library Path

⇒ Prior to specifying targets, you need to specify where resources can be found for said target.

{header files & libraries}

@ include_directories()

```
include_directories(include ${catkin_INCLUDE
                            _DIRS})
```

@ link_directories()

↳ This is not recommended.

```
link_directories(~/my_libs)
```

Executable Targets

⇒ To specify an executable target that must be built, we must use the add_executable() CMake function.

Example:

add_executable (my_Program src/main.cpp src/Sum_file.cpp)

Library Target

⇒ The add_library ( ) CMake function is used to specify libraries to build.

⇒ By default catkin builds shared libraries.

Example

add_library (${PROJECT_NAME} ${${PROJECT_NAME}_SRCS})

target_link_libraries

- Use the target_link_libraries ( ) function to specify which libraries an executable target link against.

target_link_libraries (<executable Target Name> <lib1> <lib2> ... <libn>)

★ Messages, Services and Action Targets

⇒ Message (.msg), Services (.srv) and action (.action) files in ROS require a special preprocessor build step before being built and used by ROS package.

⇒ The point of these macros is to generate programming language-specific files so that one can utilize it in their programming language of choice.

⇒ There are three macros provided to handle messages, services, and actions respectively:

```
add_message_files
add_service_files
add_action_files
```

⇒ These macros must be followed by a call to the macro that invokes generation:

```
generate_message()
```