ROS2 Basics in 5 days (C++)

Unit 1 Introduction to the Course

- Summary -

Estimated time to completion: **10 minutes**

This unit is an introduction to the **ROS2 Basics in 5 Days** Course. You'll have a quick preview of the contents you are going to cover during the course, and you will also view a practical demo.

- End of Summary -

1.1 What's this course about?

The goal of ROS2 is to overcome the limitations of ROS1, putting special attention to the creation of robotics products. For instance, ROS2 includes realtime features, embedded security and doesn't require a *roscore* (relies on DDS communication).
The goal of this course is to introduce you to the basic concepts that you need to know in order to start working with ROS2. During the course, we will try to bypass all the unnecessary noise and focus on the main things you need to know in order to learn to use ROS2. And in particular, we will focus on practice. So... what do you say? Are you in?

1.2 Let's start practising!

With the proper introductions made, it is time to actually start. And... as we always do in the Robot Ignite Academy, let's start with practice! In the following example, you will be using a simulated MARA robot, developed by Erle Robotics, which is running in ROS2. So... let's go!

- Demo -

a) First of all, you will need to source ROS2 in order to be able to execute ROS2 commands.

Execute in Shell #1

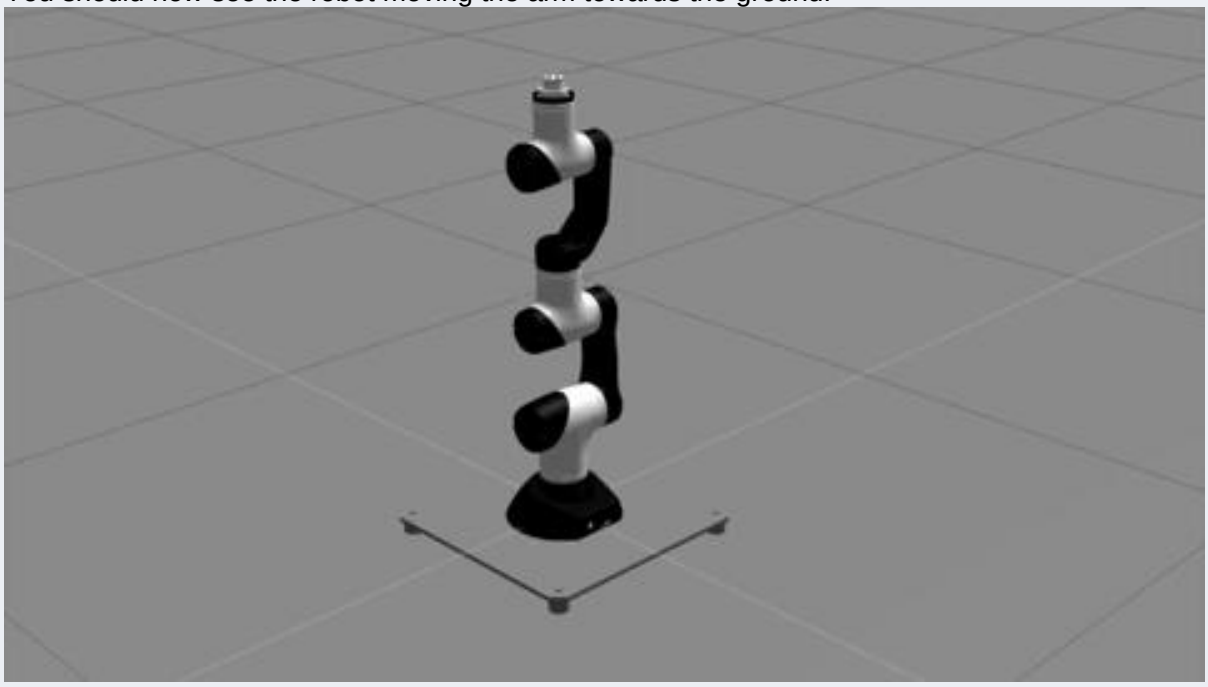
```
source .bashrc_ros2
```

b) Now, let's execute a simple ROS2 program that will execute a motion on our simulated MARA robot.

Execute in Shell #1

```
ros2 run mara_minimal_publisher mara_minimal_publisher_v1.py
```

You should now see the robot moving the arm towards the ground.



In this example, you've basically launched a ROS2 program. In this program, you are publishing some messages into certain topics, which tell the arm to move up and down continuously. Don't worry if you don't know what I'm talking about yet! I promise you that by the end of this course, you will perfectly understand what is going on behind the scenes!

Whenever you want to stop the arm's movement, you can just click **Ctrl + C** in your keyboard.

- End of Demo -

1.3 What will you learn with this course?

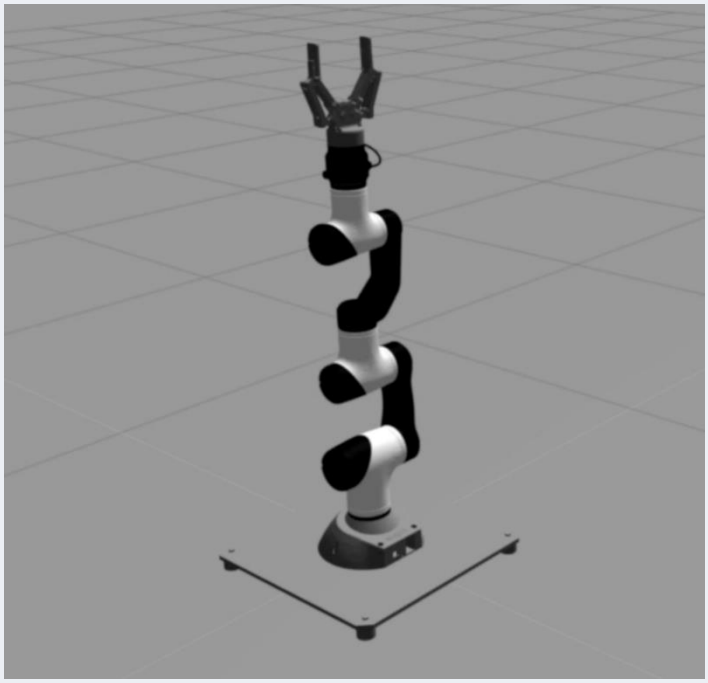
Basically, during this course, you will address the following topics:

- Basic Concepts of ROS2: Packages, Launch Files, Nodes, Client Libraries, etc...
- How to work with ROS1 Bridge
- How Topics work: Publishers and Subscribers
- What is Node Composition and how to create Components in ROS2.
- How Services work: Clients and Servers
- How Actions work: Clients and Servers
- Basic Debugging Tools: Logging system, RViz2.

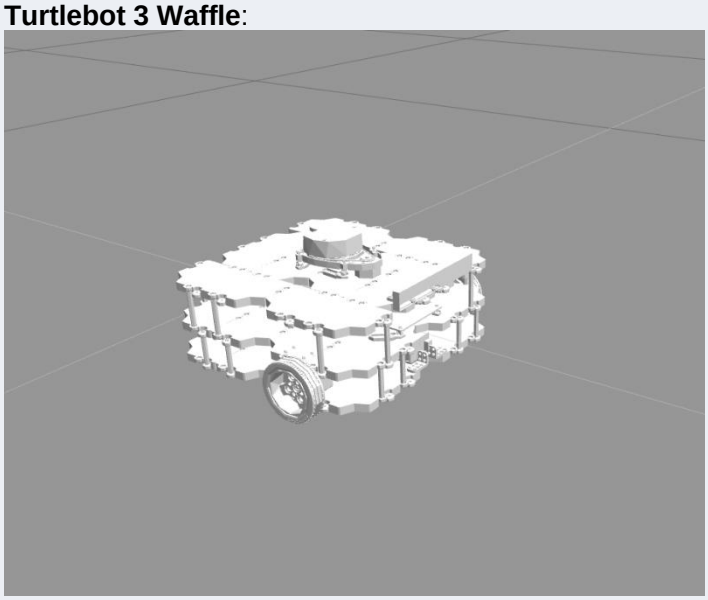
1.4 How will you learn all this?

You will learn through hands-on experience from day one! During the course, you will work with the following simulated robots.

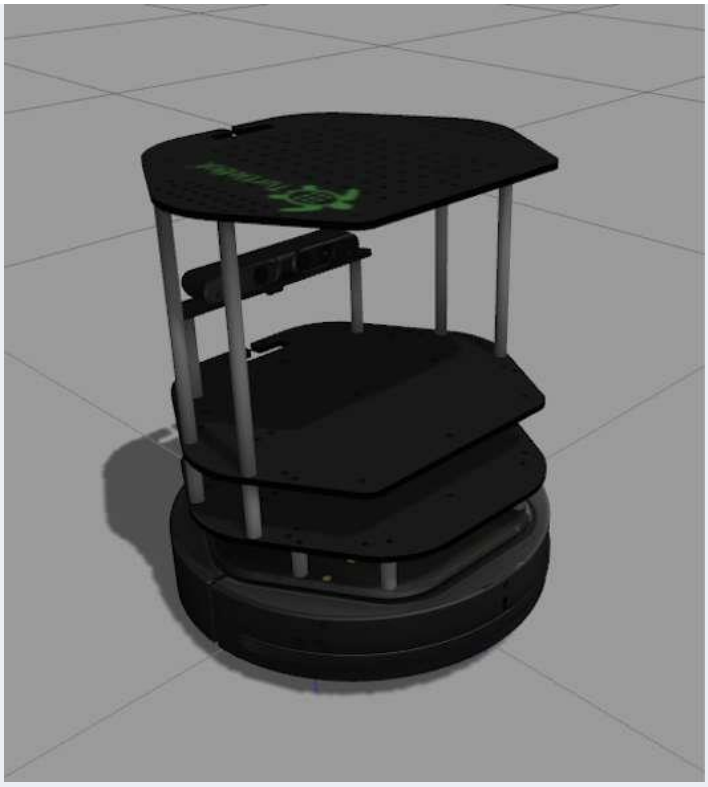
MARA robot:



Turtlebot 3 Waffle:

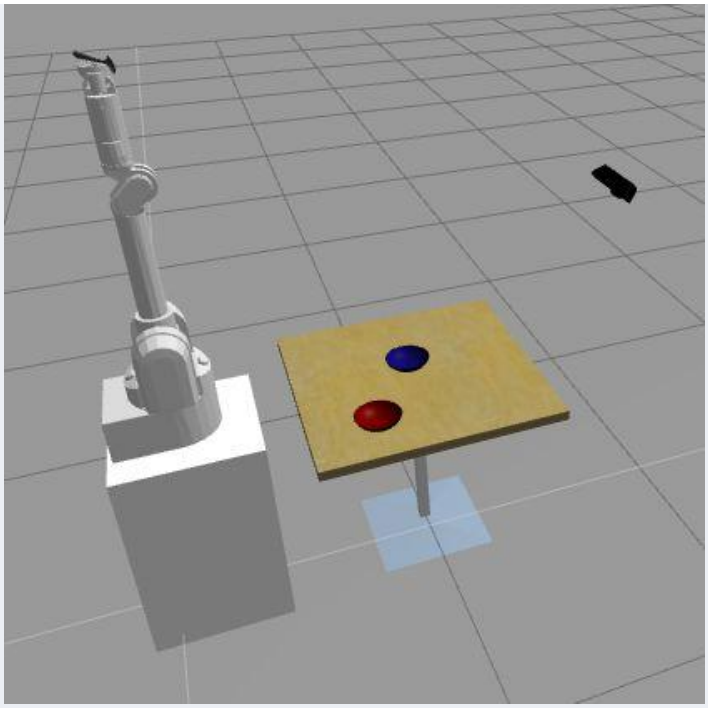


Turtlebot 2:



BB8:

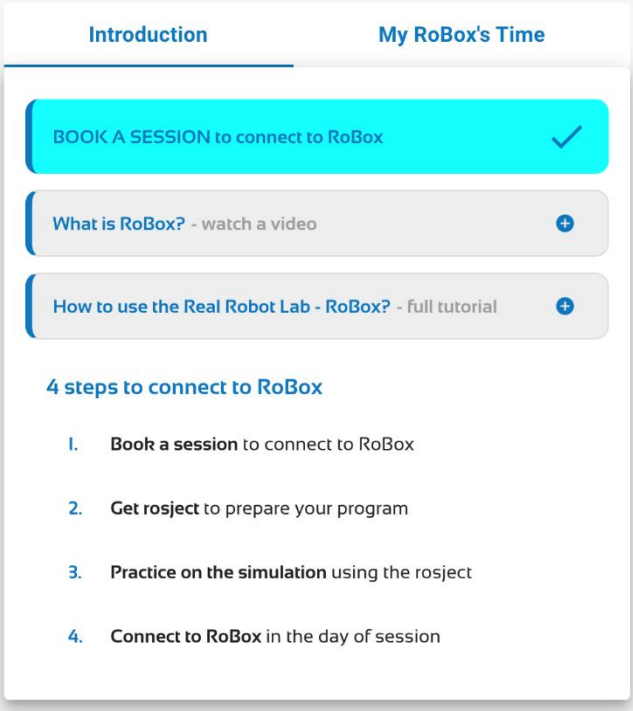
Wam Arm:

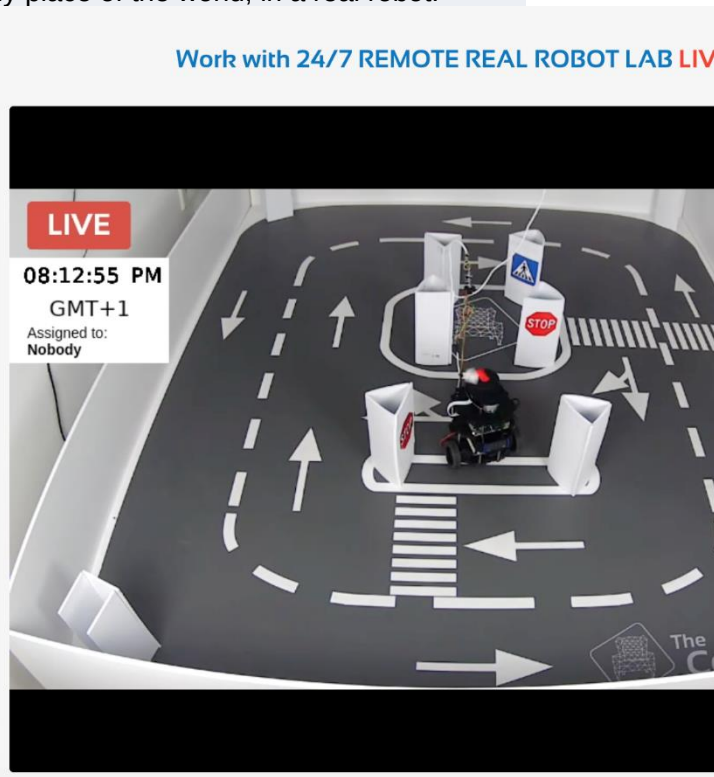


1.5 Apply what you learnt to a Real Robot Project

As you master the different concepts of ROS2, you will have to apply everything you learn during the course in a complete robot project. And I'm not talking about any kind of robot project, but a project based on a **real robot** which is located in our facilities in Barcelona.

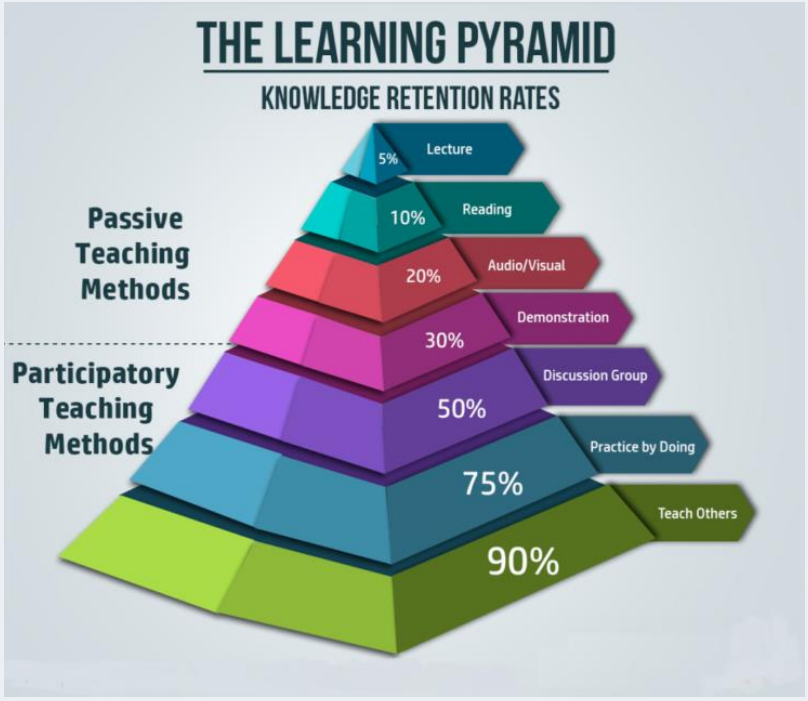
For this purpose, we will be using the [Real Robot Lab] tool. This amazing tool will allow you to remotely control and run your ROS2 programs, from any place of the world, in a real robot.





You will get introduced into the real robot project as you advance through the different units of the course.

But that's not all! After you complete the real robot project, you will have the chance to do a **live presentation** of your project. And why do we want you to show and explain your project in a live session? Because at The Construct, we strongly believe that this is the most efficient and challenging way of learning and proving your knowledge on a subjet.



1.6 Get a certificate

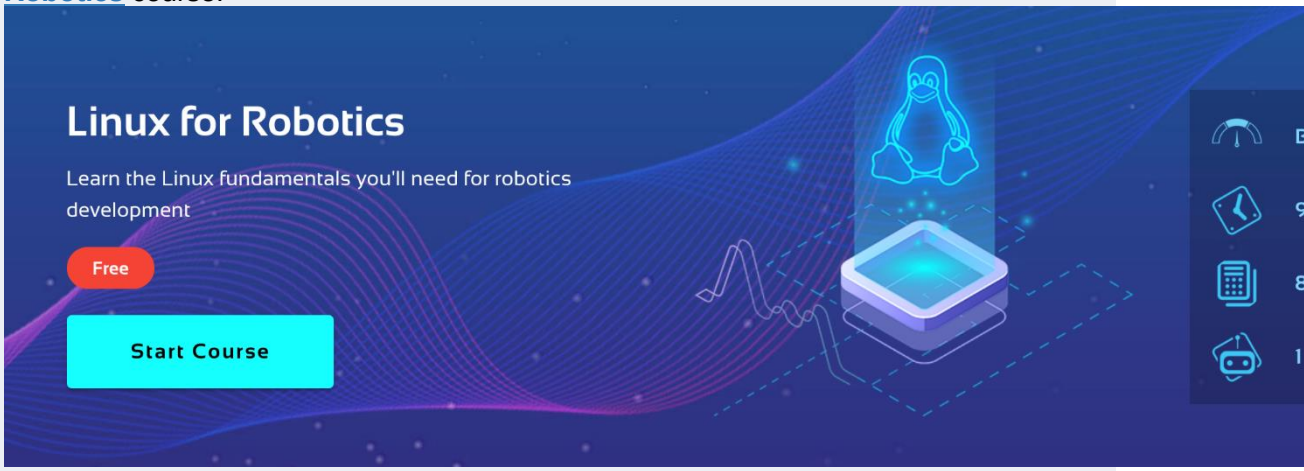
Upon completion of the course, you will be get the chance to earn a certificate proving your knowledge on ROS2 Basics. In order to earn the certificate, you will have to successfully complete the following tasks:

- Successfully complete the live presentation of your project.

1.7 Minimum requirements for the course

You need to know the following topics:

1. **Basic knowledge of Linux**. If you don't know Linux, take first our fully free [Linux For Robotics] course.



2. **Knowledge about how to create programs in C++**. If you don't know C++, take first our fully free [C++ For Robotics] course.

If you don't know how to program in C++ then we recommend you to not attempt to do this course, and instead go for the **ROS2 Basics (Python)** course.

1.8 Special Thanks

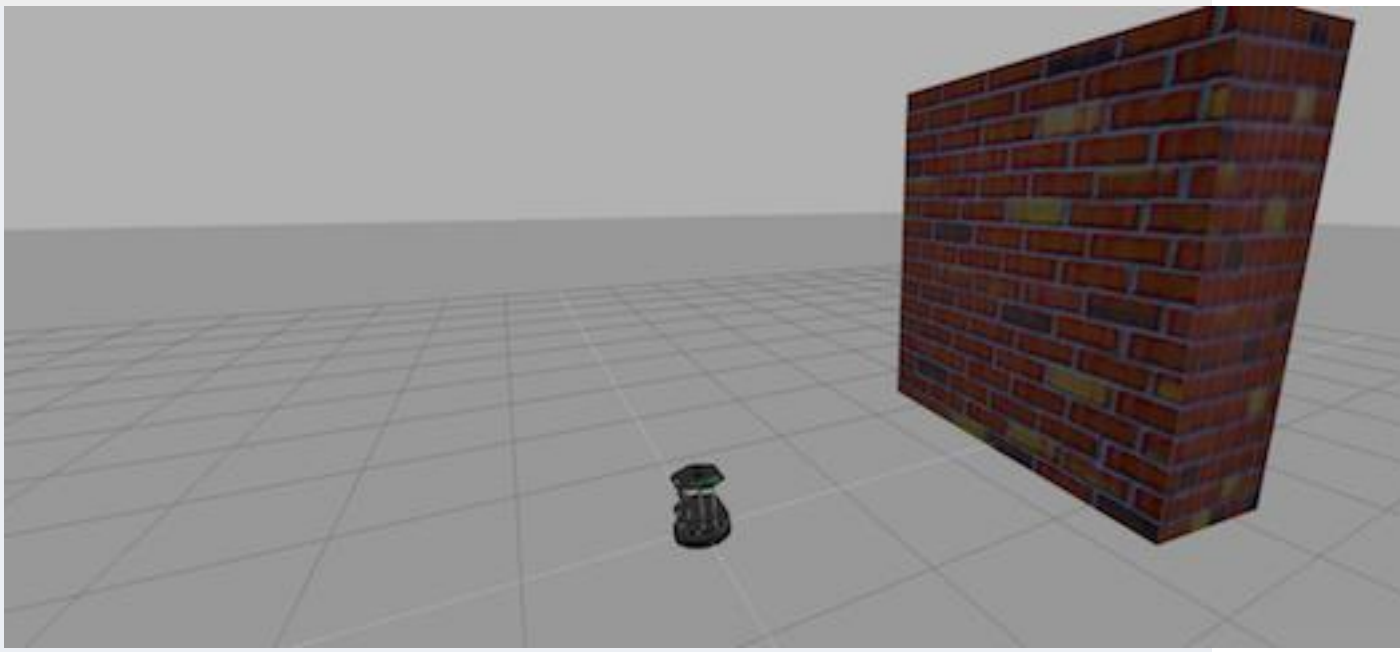
- To our friends at Erle Robotics, who have shared with us their amazing ROS2 MARA Gazebo simulation, one of the first simulations fully available in ROS2.

Erle Robotics Official Page: https://acutronicrobotics.com/

ROS2 MARA Simulation: https://github.com/AcutronicRobotics/MARA

- This course also wouldn't have been possible without the knowledge and work of the ROS Community, OSRF, and Gazebo Team.

ROS2 Basics in 5 days (C++)



Unit 2 Basic Concepts

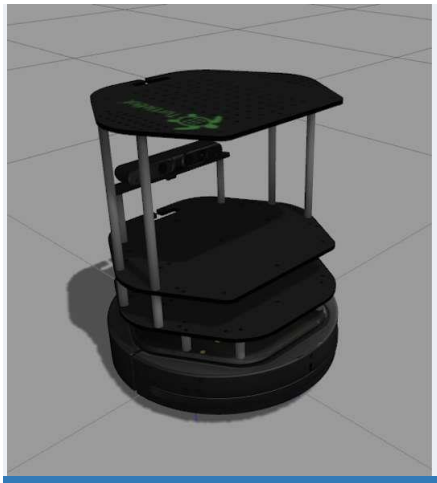
- Summary -

Estimated time to completion: 1.5 hours

Simulated robot: Turtlebot 2

What will you learn with this unit?

- How to structure and launch ROS2 programs (packages and launch files)
- How to create basic ROS2 programs (C++ based)
- Basic ROS2 concepts: Nodes, Client Libraries, etc.



- End of Summary -

2.1 What is ROS2?

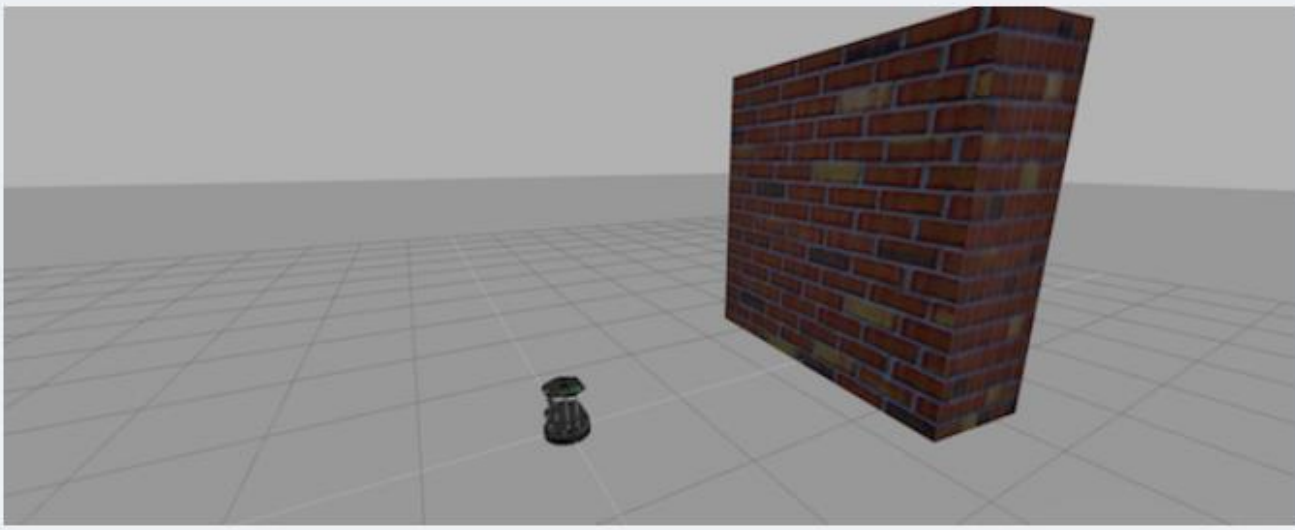
This is probably the question that has brought you all here. Well, let me tell you that you are still not prepared to understand the answer to this question, so... let's get some work done first.

2.1.1 Move a Robot with ROS2

On the right corner of the screen, you have your first simulated robot: the Turtlebot 2 robot against a large wall.

ROS2 BASICS IN 5 DAYS

Unit 1: Basic Concepts



Estimated time to completion: 1.5 hours

Simulated robot: Turtlebot 2

What will you learn with this unit?

Let's move that robot!

How do you move the Turtlebot?

The easiest way is by executing an existing ROS2 program to control the robot. A ROS2 program is executed by using some special files called **executables**, which are generated on compilation. You will see more on compilation later on in the chapter.

Since a previously-made ROS2 program (executable) already exists in this computer that allows you to move the robot using the keyboard, let's launch that ROS2 program to teleoperate the robot.

- Example 2.1 -

Before attempting to launch the program, we need to do some setup. Do not worry about the meaning of the following commands, you will gain the knowledge along this course. Now, just execute the following commands in WebShell #1 in order to start the **ROS1 Bridge** (more about *ROS1 Bridge* below).

Execute in Shell #1

```
source ~/.bashrc_bridge
```

You may get the following message on the shell:

```
ROS_DISTRO was set to 'noetic' before. Please make sure that the environment does not mix paths from differ
ent distributions.
ROS_DISTRO was set to 'foxy' before. Please make sure that the environment does not mix paths from differen
t distributions.
```

Do not worry, everything is ok. Continue with the next commands:

Execute in Shell #1

```
ros2 run ros1_bridge dynamic_bridge
```

You will get a lot of messages of the type:

```
Created 2 to 1 bridge for service /gazebo/reset_world
```

Everything is ok. Now, execute the following commands in WebShell #2:

Execute in Shell #2

```
source /opt/ros/foxy/setup.bash
ros2 run teleop_twist_keyboard teleop_twist_keyboard
```

Shell #2 Output

```
Control Your Turtlebot!
---------------------------
Moving around:
   u    i    o
   j    k    l
   m    ,    .

q/z : increase/decrease max speeds by 10%
w/x : increase/decrease only linear speed by 10%
e/c : increase/decrease only angular speed by 10%
space key, k : force stop
anything else : stop smoothly

CTRL-C to quit
```

CONGRATULATIONS! You just launched your first ROS2 program!

Now, you can use the keys indicated in the WebShell Output to move the robot around. The basic keys are the following:

i	**Move forward**
,	**Move backward**
j	**Turn left**
l	**Turn right**
k	**Stop**
q z	**Increase / Decrease Speed**

Try it!! Move the robot around using the keyboard.

IMPORTANT: remember that in order the keys to take effect, you need to have the *focus* of the terminal (shell) where you launched the program. You know you have the focus because the cursor is white and blinking.

When you're done, you can press **Ctrl+C** to stop the execution of the program (remember to have the *focus*).
ros2 is the keyword used for all the ROS2 commands. For launching programs, you will basically have two options:

- Launch the ROS2 program by directly running the **executable file**.
- Launch the ROS2 program by starting a **launch file**.

For directly running an executable file, the structure of the command goes as follows:

```
ros2 run <package_name> <executable_file>
```

As you can see, that command has two parameters: the first one is **the name of the package** that contains the executable file, and the second one is **the name of the executable file** itself (which is stored inside the package).
For using a launch file, the structure of the command would go as follows:

```
ros2 launch <package_name> <launch_file>
```

As you can see, this command also has two parameters: the first one is **the name of the package** that contains the launch file, and the second one is **the name of the launch file** itself (which is stored inside the package).

- End of Example 2.1 -

2.2 ROS1 Bridge

In the previous Demo, you had to start a **ROS1 Bridge** node in order to be able to control the robot. But... why?
As you already know, ROS2 is very young in comparison with its older brother ROS1. Because of this, there are still many packages and simulations that are not yet available for ROS2. Thankfully, we have the ROS1 Bridge to fill in these gaps.
Basically, the *ROS1 Bridge* provides a network bridge that enables the exchange of messages between ROS1 and ROS2. This way, we are able to use packages or simulations that are made for ROS1, in ROS2.

Anyway, don't worry too much about this right now. You will learn more about the ROS1 Bridge in the following chapter.

2.3 Now... what's a package?

ROS uses **packages** to organize its programs. You can think of a package as **all the files that a specific ROS program contains**; all its cpp files, python files, configuration files, compilation files, launch files, and parameters files.

All those files in the package are organized with the following structure:

- **launch** folder: Contains launch files
- **src** folder: Source files (cpp, python)
- **CMakeLists.txt**: List of cmake rules for compilation
- **package.xml**: Package information and dependencies

Every ROS program that you want to execute is organized in a package.
Every ROS program that you create will have to be organized in a package.
Packages are the main organization system of ROS programs.

2.4 Create a package

Until now we've been checking the structure of an already-built package... but now, let's create one ourselves.

When we want to create packages, we need to work in a very specific ROS workspace, which is known as ***ROS workspace***. The ROS workspace is the directory in your hard disk where your own **ROS2 packages must reside** in order to be usable by ROS2. Usually, the ***ROS workspace*** directory is called ***ros2_ws***.

- Example 2.2 -

First of all, let's source ROS2 on our Shell, so that we can use the ROS2 command line tools.

Execute in Shell #1

```
source /opt/ros/foxy/setup.bash
```

Now, go to the **ros2_ws** in your webshell.

Execute in Shell #1

```
cd ~/ros2_ws/
pwd
```

Shell #1 Output

```
user ~ $ pwd
/home/user/ros2_ws
```

Inside this workspace, there is a directory called **src**. This folder will contain all the packages created. Every time you want to create a package, you have to be in this directory (**ros2_ws/src**).Type into your webshell **cd src** in order to move to the source directory.

Execute in Shell #1

```
cd src
```

Now, we are ready to create our first package! In order to create a package, type into your webshell:

Execute in Shell #1

```
ros2 pkg create my_package --build-type ament_cmake --dependencies rclcpp
```

This will create inside our "src" directory a new package with some files in it. We'll check this later. Now, let's see how this command is built:

```
ros2 pkg create <package_name> --build-type ament_cmake --dependencies <package_dependecies>
```

The **package_name** is the name of the package you want to create, and the **package_dependencies** are the names of other ROS packages that your package depends on.

- End of Example 2.2 -

- Example 2.3 -

In order to check that our package has been created successfully, we can use some ROS commands related to packages. For example, let's type:

Execute in Shell #1

```
ros2 pkg list
ros2 pkg list | grep my_package
```

ros2 pkg list: Gives you a list with all of the packages in your ROS system.
ros2 pkg list | grep my_package: Filters, from all of the packages located in the ROS system, the package named *my_package*.

You can also see the package created and its contents by just opening it through the IDE (similar to {Figure 2.1})

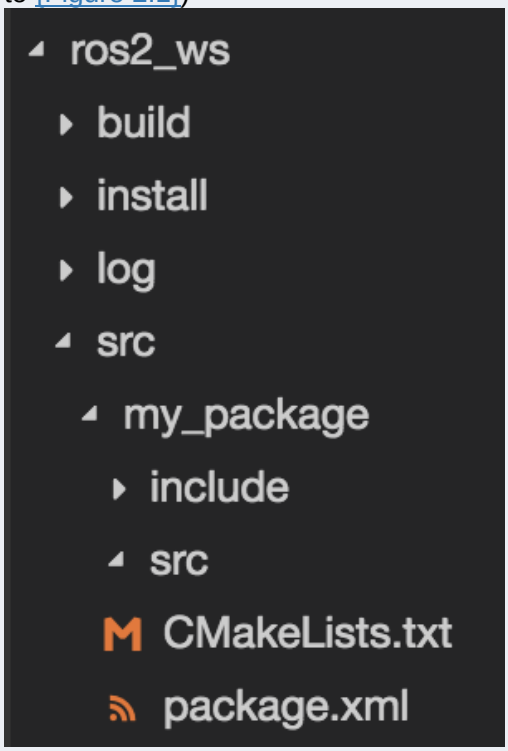


Fig.2.1 - IDE created package my_package

So... what's happening? Can't see your package on the list? Well, don't worry. That's totally normal. In order to see your new package on the package list, you will need to compile it first. So, let's move on to the next section to see how to compile a package.

- End of Example 2.3 -

2.5 Compile a package

When you create a package, you will need to compile it in order to make it work. The command used by ROS2 to compile is the next one:

```
colcon build
```

This command will compile your whole **src** directory, and **it needs to be issued in your *ros2_ws* directory in order to work. This is MANDATORY.**

- Example 2.4 -

Go to your **ros2_ws** directory and compile your source folder. You can do this by typing:

Execute in Shell #1

```
cd ~/ros2_ws
colcon build
```

Sometimes (for example, in large projects) you will not want to compile all of your packages, but just the one(s) where you've made changes. You can do this with the following command:

```
colcon build --packages-select <package_name>
```

This command will only compile the packages specified and their dependencies.
Try to compile your package named my_package with this command.
Execute in Shell #1

```
colcon build --packages-select my_package
```

When compilation ends, you will need to source your workspace. You can do that with the following command:
Execute in Shell #1

```
source ~/ros2_ws/install/setup.bash
```

- End of Example 2.4 -

2.6 And... what's a launch file?

We've seen that ROS2 can use launch files to execute programs. But... how do they work? Let's have a look.

- Example 2.5 -

In Example 2.1, you used the command ***ros2 run*** in order to start the **teleop_twist_keyboard** node. But you've also seen that you can start nodes by using what we know as ***launch files***.
But... how do they work? Let's have a look at an example. For instance, if we wanted to start the **teleop_twist_keyboard** node using a launch file, we would have to create something similar to the following Python script.

teleop_twist_keyboard.launch.py

```
from launch import LaunchDescription
import launch_ros.actions


def generate_launch_description():
    return LaunchDescription([
        launch_ros.actions.Node(
            package='teleop_twist_keyboard', executable='teleop_twist_keyboard', output='screen'),
    ])
```

As you can see, the launch file structure is quite simple. First, we import some modules from the **launch** and **launch_ros** packages.

```
from launch import LaunchDescription
import launch_ros.actions
```

Next, we define a function that will return a **LaunchDescription** object.

```
def generate_launch_description():
    return LaunchDescription([
        launch_ros.actions.Node(
            package='teleop_twist_keyboard', executable='teleop_twist_keyboard', output='screen'),
    ])
```

Within the LaunchDescription object, we generate a Node where we will fill up the following parameters:

1. **package='package_name'** # Name of the package that contains the code of the ROS program to execute
2. **executable='cpp_executable_name'** # Name of the cpp executable file that we want to execute

3. **output='type_of_output'** # Through which channel you will print the output of the program

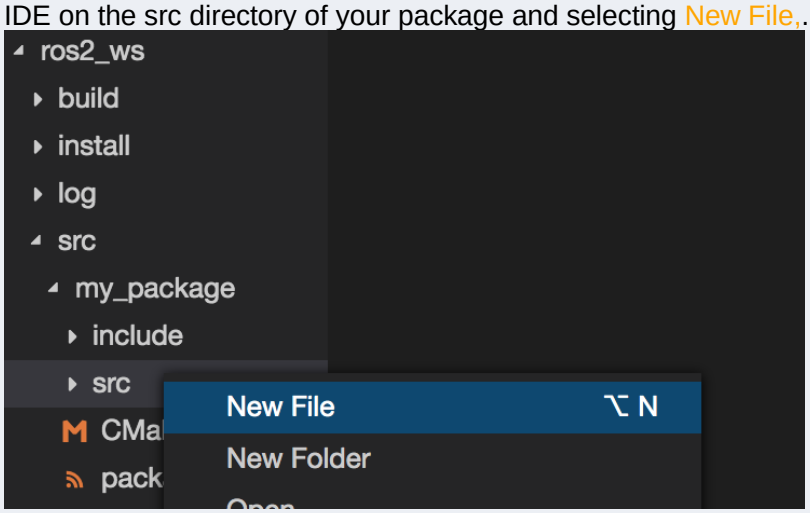
- End of Example 2.5 -

2.7 My first ROS program

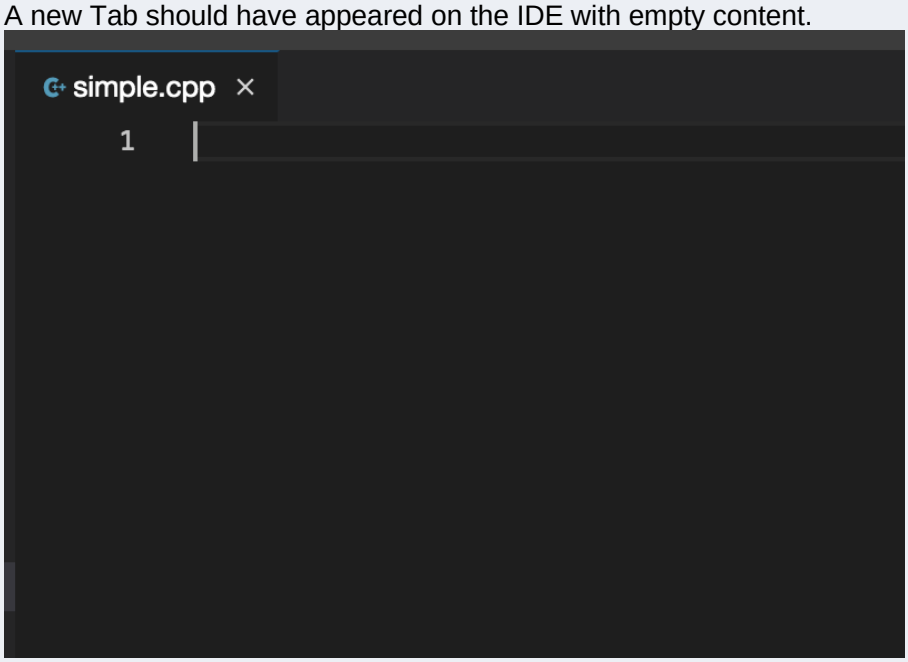
At this point, you should have your first package created... but now, you need to do something with it! Let's do our first ROS2 program!

- Example 2.6 -

1- Create a C++ file that will be executed in the **src** directory in my_package. For this exercise, just copy this simple C++ code simple.cpp. You can create it directly by **RIGHT clicking** on the IDE on the src directory of your package and selecting New File,.



A new Tab should have appeared on the IDE with empty content.



Copy the content of simple.cpp into the new file.

simple.cpp

```cpp
#include "rclcpp/rclcpp.hpp"

int main(int argc, char * argv[])
{
  rclcpp::init(argc, argv);
  auto node = rclcpp::Node::make_shared("ObiWan");

  RCLCPP_INFO(node->get_logger(), "Help me Obi-Wan Kenobi, you're my only hope");

  rclcpp::shutdown();
  return 0;
}
```

2- Create a *launch* directory inside the package named my_package {Example 2.2}.

Execute in Shell #1

```
cd ~/ros2_ws/src/my_package
mkdir launch
```

You can also create the *launch* directory through the IDE.

3- Create a new launch file inside the launch directory.

Execute in Shell #1

```
cd ~/ros2_ws/src/my_package/launch
touch my_package_launch_file.launch.py
chmod +x my_package_launch_file.launch.py
```

You can also create it through the IDE.

4- Fill this launch file as we've previously seen in this course {Example 2.5}.

The final launch should be something similar to this: my_package_launch_file.launch

5- Modify the **CMakeLists.txt** file to generate an executable from the C++ file you have just created.

Note: This is something that is **required when working in ROS with C++**. When you finish this exercise, you'll learn more about this subject. For now, just follow the instructions below.

In the **Build** section of your **CMakeLists.txt** file, add the following lines to your **CMakeLists.txt** file, right above the **ament_package()** line.

```
add_executable(simple_node src/simple.cpp)
ament_target_dependencies(simple_node rclcpp)

install(TARGETS
  simple_node
  DESTINATION lib/${PROJECT_NAME}
)

install(DIRECTORY
  launch
  DESTINATION share/${PROJECT_NAME}/
)
```

```
21
22 if(BUILD_TESTING)
23   find_package(ament_lint_auto REQUIRED)
24   # the following line skips the linter which checks for copyrights
25   # remove the line when a copyright and license is present in all source files
26   set(ament_cmake_copyright_FOUND TRUE)
27   # the following line skips cpplint (only works in a git repo)
28   # remove the line when this package is a git repo
29   set(ament_cmake_cpplint_FOUND TRUE)
30   ament_lint_auto_find_test_dependencies()
31 endif()
32
33 add_executable(simple_node src/simple.cpp)
34 ament_target_dependencies(simple_node rclcpp)
35
36 install(TARGETS
37    simple_node
38    DESTINATION lib/${PROJECT_NAME}
39  )
40
41 install(DIRECTORY
42   launch
43   DESTINATION share/${PROJECT_NAME}/
44 )
45
46 ament_package()
47
```

6- Compile your package as explained previously.

Execute in Shell #1

```
cd ~/ros2_ws;
colcon build
source ~/ros2_ws/install/setup.bash
```

If everything goes fine, you should get something like this as output:

```
user:~/ros2_ws$ colcon build --symlink-install
Starting >>> my_package
Finished <<< my_package [12.7s]

Summary: 1 package finished [12.8s]
```

7- Finally, execute the roslaunch command in the WebShell to launch your program.

Execute in Shell #1

```
ros2 launch my_package my_package_launch_file.launch.py
```

- End of Example 2.6 -

- Expected Result for Example 2.6 -

You should see Leia's quote among the output of the roslaunch command.

Shell #1 Output

```
[INFO] [launch]: All log files can be found below /home/user/.ros/log/2021-01-07-19-07-18-415200-1_xterm-1724
[INFO] [launch]: Default logging verbosity is set to INFO
[INFO] [simple_node-1]: process started with pid [1726]
[simple_node-1] [INFO] [1610046438.644830125] [ObiWan]: Help me Obi-Wan Kenobi, you're my only hope
[INFO] [simple_node-1]: process has finished cleanly [pid 1726]
```

- End of Expected Result -

C++ Program: simple.cpp

```
#include "rclcpp/rclcpp.hpp"
```

```
int main(int argc, char * argv[])
{
  rclcpp::init(argc, argv);
  auto node = rclcpp::Node::make_shared("ObiWan");

  RCLCPP_INFO(node->get_logger(), "Help me Obi-Wan Kenobi, you're my only hope");

  rclcpp::shutdown();
  return 0;
}
```

You may be wondering what this whole code means, right? Well, let's explain it line by line:

```
// Here we are including all the headers necessary to use the most common public pieces of the ROS system
// In this case we use the rclcpp client library, which provides a C++ Api for interacting with ROS
// Always, when we create a new C++ file, we will need to add this include:
#include "rclcpp/rclcpp.hpp"

// We start the main C++ program
int main(int argc, char * argv[])
{
  // We initiate the rclcpp client library
  rclcpp::init(argc, argv);
  // We initiate a ROS node called ObiWan
  auto node = rclcpp::Node::make_shared("ObiWan");

  // This is the same as a print in ROS
  RCLCPP_INFO(node->get_logger(), "Help me Obi-Wan Kenobi, you're my only hope");

  // We shutdown the rclcpp client library
  rclcpp::shutdown();
  // We end our program
  return 0;
}
```

NOTE: If you create your C++ file from the shell, it may happen that it's created without execution permissions. If this happens, ROS won't be able to find it. If this is the case for you, you can give execution permissions to the file by typing the next command: ***chmod +x name_of_the_file.cpp***

END C++ Program: simple.cpp

Launch File: my_package_launch_file.launch.py

You should have something similar to this in your my_package_launch_file.launch:

Note: Keep in mind that in the example below, the C++ executable name in the attribute **node_executable** is named **simple_node**. So, if you have named your C++ executable with a different name, this will be different.

```
"""Launch file example"""

from launch import LaunchDescription
import launch_ros.actions


def generate_launch_description():
```

```
    return LaunchDescription([
        launch_ros.actions.Node(
            package='my_package', executable='simple_node', output='screen'),
    ])
```

END Launch File: my_package_launch_file.launch.py

2.8 Modifying the CMakeLists.txt file

When coding with C++, it will be necessary to create binaries (executables) of your programs in order to be able to execute them. For that, you will need to modify the **CMakeLists.txt** file of your package, in order to indicate that you want to create an executable of your C++ file.

To do this, you need to add some lines into your **CMakeLists.txt** file. In fact, these lines are already in the file, but they are commented. You can also find them, and uncomment them. Whichever you prefer.

In the previous exercise, you had the following lines:

```
add_executable(simple_node src/simple.cpp)
ament_target_dependencies(simple_node rclcpp)


install(TARGETS
  simple_node
  DESTINATION lib/${PROJECT_NAME}
)

# Install launch files.
install(DIRECTORY
  launch
  DESTINATION share/${PROJECT_NAME}/
)
```

But... what do these lines of code do exactly? Well, basically they do the following:

```
add_executable(simple_node src/simple.cpp)
```

This line **generates an executable from the simple.cpp file**, which is in the src folder of your package. This executable will be called **simple_node**.

```
ament_target_dependencies(simple_node rclcpp)
```

This line adds all the ament target dependencies of the executable.

```
install(TARGETS
  simple_node
  DESTINATION lib/${PROJECT_NAME}
)
```

This snippet will currently install our node (**simple_node**) into our install space inside the ROS2 workspace. So, this executable **will be placed into the package directory of your install space**, which is located, by default, at **~/ros2_ws/install/<package name>/lib**.

```
install(DIRECTORY
  launch
  DESTINATION share/${PROJECT_NAME}/
)
```

Finally, this code snippet will install the launch files into tne install space so that they can be executed using the ***ros2 launch*** expression.

2.9 ROS2 Nodes

You've initiated a node in the previous code but... what's a node? ROS nodes are basically programs made in ROS. The ROS command to see what nodes are actually running in a computer is:

```
ros2 node list
```

- Example 2.7 -

Type this command in a new shell and look for the node you've just initiated (ObiWan).

Execute in Shell #1

```
ros2 node list
```

You can't find it? I know you can't. That's because the node is killed when the C++ program ends.
Let's change that.

Update your C++ file simple.cpp with the following code:

C++ Program: simple_loop.cpp

```
#include "rclcpp/rclcpp.hpp"

int main(int argc, char * argv[])
{
  rclcpp::init(argc, argv);
  auto node = rclcpp::Node::make_shared("ObiWan");
  // We create a Rate object of 2Hz
  rclcpp::WallRate loop_rate(2);

  // Endless loop until Ctrl + C
  while (rclcpp::ok()) {

    RCLCPP_INFO(node->get_logger(), "Help me Obi-Wan Kenobi, you're my only hope");
    rclcpp::spin_some(node);
    // We sleep the needed time to maintain the Rate fixed above
    loop_rate.sleep();

  }
  rclcpp::shutdown();
  return 0;
}
// This program creates an endless loop that repeats itself 2 times per second (2Hz) until somebody presses Ctrl + C
// in the Shell
```

END C++ Program: simple_loop.cpp

Launch your program again using the roslaunch command.

Execute in Shell #1

```
ros2 launch my_package my_package_launch_file.launch.py
```

Now, try again in another WebShell:

Execute in Shell #2

```
source /opt/ros/foxy/setup.bash
ros2 node list
```

Can you see your node now?

Shell #2 Output

```
user:~$ ros2 node list
/ObiWan
```

In order to see information about our node, we can use the next command:

```
ros2 node info <node_name>
```

This command will show us information about all the connections that our Node has.

Execute in Shell #2

```
ros2 node info /ObiWan
```

Shell #2 Output

```
user:~$ ros2 node info /ObiWan
/ObiWan
  Subscribers:
    /parameter_events: rcl_interfaces/msg/ParameterEvent
  Publishers:
    /parameter_events: rcl_interfaces/msg/ParameterEvent
    /rosout: rcl_interfaces/msg/Log
  Service Servers:
    /ObiWan/describe_parameters: rcl_interfaces/srv/DescribeParameters
    /ObiWan/get_parameter_types: rcl_interfaces/srv/GetParameterTypes
    /ObiWan/get_parameters: rcl_interfaces/srv/GetParameters
    /ObiWan/list_parameters: rcl_interfaces/srv/ListParameters
    /ObiWan/set_parameters: rcl_interfaces/srv/SetParameters
    /ObiWan/set_parameters_atomically: rcl_interfaces/srv/SetParametersAtomically
  Service Clients:

  Action Servers:

  Action Clients:
```

For now, don't worry about the output of the command. You will understand more while going through the next tutorial.

- End of Example 2.7 -

2.10 Client Libraries

In the previous exercise, you were using the **rclcpp** client library. But what is this exactly? Basically, ROS client libraries allow nodes written in different programming languages to communicate. There is a core ROS client library (RCL) that implements the common functionality needed for the ROS APIs of different languages. This makes it so that language-specific client libraries are easier to write and they have more consistent behavior.

The following client libraries are currently maintained by the ROS2 team:

- **rclcpp** = C++ client library
- **rclpy** = Python client library

Additionally, other client libraries have been developed by the ROS community. You can check out the following article for more details: https://index.ros.org/doc/ros2/ROS-2-Client-Libraries/

2.11 Environment Variables

ROS uses a set of Linux system environment variables in order to work properly. You can check these variables by typing:

Execute in Shell #1

```
export | grep ROS
```

Shell #1 Output

```
user:~$ export | grep ROS
declare -x ROSLISP_PACKAGE_DIRECTORIES="/home/user/catkin_ws/devel/share/common-lisp:/home/simulations/public_sim_ws/devel/share/common-lisp"
declare -x ROS_DISTRO="foxy"
declare -x ROS_ETC_DIR="/opt/ros/noetic/etc/ros"
declare -x ROS_HOSTNAME="1_xterm"
declare -x ROS_IP=""
declare -x ROS_LOCALHOST_ONLY="0"
declare -x ROS_MASTER_URI="http://1_simulation:11311"
declare -x ROS_PACKAGE_PATH="/home/user/catkin_ws/src:/home/simulations/public_sim_ws/src:/opt/ros/noetic/share"
declare -x ROS_PYTHON_VERSION="3"
declare -x ROS_ROOT="/opt/ros/noetic/share/ros"
declare -x ROS_VERSION="2"
declare -x SLOT_ROSBRIDGE_PORT="20001"
```

The most important variables are the **ROS_MASTER_URI** and the **ROS_DISTRO**.

ROS_MASTER_URI -> Contains the url where the ROS Core **is** being executed.

ROS_DISTRO -> Contains the ROS distribution that you are currently using.

- Notes -

NOTE 1: Since ROS2 doesn't has a roscore, the **ROS_MASTER_URI** is used here to point to the roscore running in ROS1, when communicating to it using the ROS1 Bridge.
NOTE 2: At the platform you are using for this course, we have created an alias to display the environment variables of ROS. This alias is **rosenv**. By typing this on your shell, you'll get a list of ROS environment variables. It is important that you know that this is **not an official ROS command**, so you can only use it while working on this platform.

- End of Notes -

2.12 So now... what is ROS2?

ROS2 is basically the framework that allows us to do all that we showed you throughout this chapter. It provides the background to manage all these processes and communications between them... and much, much more!! In this tutorial, you've just scratched the surface of ROS2, the basic concepts. ROS2 is an extremely powerful tool. If you dive into our courses, you'll learn much more about ROS2 and you'll find yourself able to do almost anything with your robots!

ROS2 Basics in 5 days (C++) ¶

Unit 3 ROS1 Bridge ¶

- Summary –

Estimated time to completion: 6 hours

What will you learn with this unit?

- Setup of ROS1 Bridge
- Basic Examples

- Exercises with simulation
- - End of Summary -

3.1 Setup of ROS1 Bridge

At the time of the creation of this course notebook, ROS2 Dashing was not a 100% replacement for ROS1, and therefore, it will have to coexists with ROS1 systems. So, we need some method to connect ROS2 systems with ROS1 systems. This is particularly true in Gazebo Simulations. Although there are Gazebo-Simulations already prepared for ROS2, like the **MARA Robot Arm** that you will use in this course, made by the team of ErleRobotics, there is not much else out there. So, you need some way of launching already existing simulations using ROS1, but able to communicate with ROS2.

This is where **ROS1-Bridge** comes into play.
Ros1-Bridge connects messages from ROS1 and ROS2. The prebuilt version supports some messages, and most of the ROS core messages used.

This mapping between messages from ROS1-ROS2 is defined at compile time through **yaml** files exported in the **package.xml**. More info on this Mapping Topic: https://github.com/ros2/ros1_bridge/blob/master/doc/index.rst

But in this course, we will concentrate on using only the default messages for the time being.

3.1.1 ROS1-Bridge sourcing

So, for this first example, we are going to just reproduce the standard tests for ROS1-Bridge. This way, you will get the hang of how this system works.
The first thing to do is to create a custom bridge bashrc configuration to be able to source both the ROS1 and ROS2 systems that ROS1-Bridge needs.

In this case, you already have it created in your workspace.

Execute in Shell #1

```
cat /home/user/.bashrc_bridge
```

```
**Setup: .bashrc_bridge**
```

```
source /opt/ros/noetic/setup.bash
source /opt/ros/foxy/local_setup.bash
source /home/simulations/ros2_sims_ws/install/setup.bash
source /home/user/catkin_ws/devel/setup.bash
source /home/user/ros2_ws/install/local_setup.bash
```

```
**END Setup: .bashrc_bridge**
```

ROS1-Bridge is based on several concepts, but the most basic one is the fact that the shell where you launch **ROS1-Bridge** has to have sourced, all the paths to all the messages that it has access to. This means that it has to be able to reach both ROS1 and ROS2 message definitions. That's why both **melodic** and **dashing** are sourced, to be able to reach ROS1 and ROS2 system installed packages. And also, it has to be able to reach any workspace that you might use. In this case, the **catkin_ws** for ROS1 and **ros2_ws** for ROS2.
Once **ros1_bridge** is correctly sourced, it basically checks if the topics and services in ROS1 and ROS2 have the same names, message types, and so on. If so, it connects the corresponding topics and services. If you want to know in more depth how this correspondance is done, please check out the official documentation here: https://github.com/ros2/ros1_bridge/blob/master/doc/index.rst

3.2 Basic Examples

Here we will perform the basic classical ROS1-Bridge example tests, to get the hang of ROS1-Bridge.
Important Note: Please start from fresh webshells to avoid any past sourcing. This is done by just going to another unit and back to this one, to reset all the webshells.
We will use **two extra bashrc_rosX**, one each for ROS1 and ROS2. Again, these files are **already created for you**, but we show you just in case you execute this someplace where there is none.

Setup: .bashrc_ros1

```
# ROS1
export ROS_DISTRO=noetic
source /opt/ros/$ROS_DISTRO/setup.bash
source /home/user/catkin_ws/devel/setup.bash
```

END Setup: .bashrc_ros1

Setup: .bashrc_ros2

```
# ROS2
export ROS_DISTRO=foxy
source /opt/ros/$ROS_DISTRO/setup.bash
source /home/user/ros2_ws/install/local_setup.bash
```

END Setup: .bashrc_ros2

Example 1a: Run the bridge and the example talker and listener ROS1->ROS2

For this, you will need **three** different webshells, which we will name (for future reference):

- WebShell 1: ROS1-Bridge
- WebShell 2: Talker ROS1
- WebShell 3: Listener ROS2

NOTE: In Robot Ignite Academy, you already have a **roscore** running, just because you have gazebo for ROS1 in the simulations that use ROS1, so you don't need to launch it. But in an empty local computer, you would need to do it, sourcing **ROS1** paths.

Execute in Shell #1: ROS1-Bridge

We launch ROS1-Bridge, sourcing for both ROS1 and ROS2, using the provided **.bashrc_bridge**.

```
. /home/user/.bashrc_bridge
ros2 run ros1_bridge dynamic_bridge
```

Execute in Shell #2: Talker ROS1

```
. /home/user/.bashrc_ros1
rosrun rospy_tutorials talker
```

Execute in Shell #3: Listener ROS2

```
. /home/user/.bashrc_ros2
ros2 run demo_nodes_cpp listener
```

Shell #1 Output: ROS1-Bridge

```
created 1to2 bridge for topic '/chatter' with ROS 1 type 'std_msgs/String' and ROS 2 type 'std_msgs/msg/String'
```

[INFO] [1610046945.906204653] [ros_bridge]: Passing message **from** ROS 1 std_msgs/String to ROS 2 std_msgs/msg/String (showing msg only once per type)
[INFO] [1610046945.907263375] [ros_bridge]: Passing message **from** ROS 2 rcl_interfaces/msg/Log to ROS 1 rosgraph_msgs/Log (showing msg only once per type)

Shell #2 Output: Talker ROS1

[INFO] [1610046948.716015, 187.171000]: hello world 187.171
[INFO] [1610046948.816289, 187.271000]: hello world 187.271
[INFO] [1610046948.916459, 187.371000]: hello world 187.371

Shell #3 Output: Listener ROS2

[INFO] [1610046947.311252590] [listener]: I heard: [hello world 185.771]
[INFO] [1610046947.410119110] [listener]: I heard: [hello world 185.871]
[INFO] [1610046947.511251514] [listener]: I heard: [hello world 185.971]

Example 1b: Run the bridge and the example talker and listener ROS2->ROS1

For this you will need **three** different webshells, which we will name (for future reference):

- WebShell 1: ROS1-Bridge
- WebShell 2: Listener ROS1
- WebShell 3: Talker ROS2

Execute in Shell #1: ROS1-Bridge

```
. /home/user/.bashrc_bridge
ros2 run ros1_bridge dynamic_bridge
```

Execute in Shell #2: Listener ROS1

```
. /home/user/.bashrc_ros1
rosrun roscpp_tutorials listener
```

Execute in Shell #3: Talker ROS2

```
. /home/user/.bashrc_ros2
ros2 run demo_nodes_py talker
```

Shell #1 Output: ROS1-Bridge

created 2to1 bridge **for** topic '/chatter' **with** ROS 2 type 'std_msgs/msg/String' **and** ROS 1 type ''
[INFO] [1610047054.689906443] [ros_bridge]: Passing message **from** ROS 2 std_msgs/msg/String to ROS 1 std_msgs/String (showing msg only once per type)

Shell #2 Output: Listener ROS1

[INFO] [1610047054.691867517, 292.586000000]: I heard: [Hello World: 0]
[INFO] [1610047055.667615171, 293.557000000]: I heard: [Hello World: 1]
[INFO] [1610047056.666984573, 294.553000000]: I heard: [Hello World: 2]

Shell #3 Output: Talker ROS2

[INFO] [1610047054.688922751] [talker]: Publishing: "Hello World: 0"
[INFO] [1610047055.666858792] [talker]: Publishing: "Hello World: 1"
[INFO] [1610047056.666301450] [talker]: Publishing: "Hello World: 2"

Example 2: Run the bridge for AddTwoInts service

For this you will need **four** different webshells, which we will name (for future reference):

- WebShell 1: ROS1-Bridge
- WebShell 2: add_two_ints_server ROS1
- WebShell 3: add_two_ints_client ROS2

Execute in Shell #1: ROS1-Bridge

```
# .bashrc_bridge sources everything and was used for the compilation
. /home/user/.bashrc_bridge
ros2 run ros1_bridge dynamic_bridge
```

Execute in Shell #2: add_two_ints_server ROS1

```
# .bashrc_bridge sources everything and was used for the compilation
. /home/user/.bashrc_ros1
rosrun roscpp_tutorials add_two_ints_server
```

Execute in Shell #3: add_two_ints_client ROS2

```
# .bashrc_bridge sources everything and was used for the compilation
. /home/user/.bashrc_ros2
ros2 run demo_nodes_cpp add_two_ints_client
```

Shell #1 Output: ROS1-Bridge

```
Created 2 to 1 bridge for service /add_two_ints
```

Shell #2 Output: add_two_ints_server ROS1

```
[ INFO] [1610047198.957038044, 436.150000000]: request: x=2, y=3
[ INFO] [1610047198.958696485, 436.151000000]:   sending back response: [5]
```

Shell #3 Output: add_two_ints_client ROS2

```
[INFO] [1610047198.959420838] [add_two_ints_client]: Result of add_two_ints: 5
```

3.3 Exercises with simulation

Now that we know how to move around with ROS1-Bridge, let's do more examples accessing Robot systems and simulations.
We will do the following examples:

- View the images published in ROS1-Gazebo by the Cameras of the robot with ROS2 systems.
- Tell the robot to move in a ROS1-Gazebo simulation through ROS2.
- Reset the simulation of ROS1-Gazebo through ROS2

- Example 3.1 -

View the images published in ROS1-Gazebo by the Cameras of the robot with ROS2 systems

For this, you will need **three** different webshells, which we will name (for future reference):

- WebShell 1: ROS1-Bridge
- WebShell 2: Image topic remapper ROS1, remapping the Robots Image topic to the topic /image.
- WebShell 3: show_image ROS2

Execute in Shell #1: ROS1-Bridge

```
. /home/user/.bashrc_bridge
ros2 run ros1_bridge dynamic_bridge
```

Execute in Shell #2: Image topic remapper ROS1

We need to republish the Robot's image topic into a topic that the **ros2 run image_tools showimage** can use. At the time of the creation of this course, it only supports reading from a topic called **/image**. This means that if your Robot Camera topic is called **my_robot/raw**, this topic has to be republished in the topic **/image**. And that's why we need to use

the **republish** binary from the ROS1 package **image_transport** to do just that. See the following official documentation for more info.

```
. /home/user/.bashrc_ros1
rosrun image_transport republish raw in:=/bb8/camera1/image_raw out:=/image
```

Execute in Shell #3: show_image ROS2

For the moment, this ROS2 image GUI only publishes from the topic named **/image**. This means that we will have to remap it in some way to be able to get images from any ROS1 image topic. See the source file for more details LINK. But because we did that on the previous step, and we have **ROS1-Bridge** working, it's just a matter of firing up this **showimage** and **ROS1-Bridge** will detect that ROS2 is trying to read from the **Image** topic and connect them. And because we remapped, the **image** topic will be a mirror image of the **/bb8/camera1/image_raw**.

```
. /home/user/.bashrc_ros2
ros2 run image_tools showimage
```

Shell #1 Output: ROS1-Bridge

```
created 1to2 bridge for topic '/image' with ROS 1 type 'sensor_msgs/Image' and ROS 2 type 'sensor_msgs/msg/Image'
[INFO] [1610047320.023758659] [ros_bridge]: Passing message from ROS 1 sensor_msgs/Image to ROS 2 sensor_msgs/msg/Image (showing msg only once per type)
```

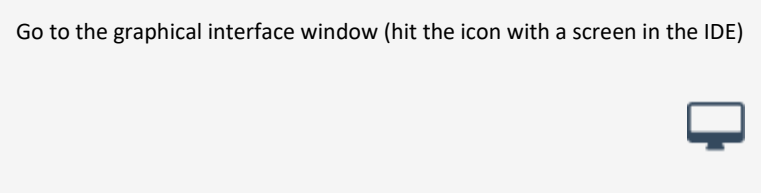
Shell #2 Output: Image topic remapper ROS1

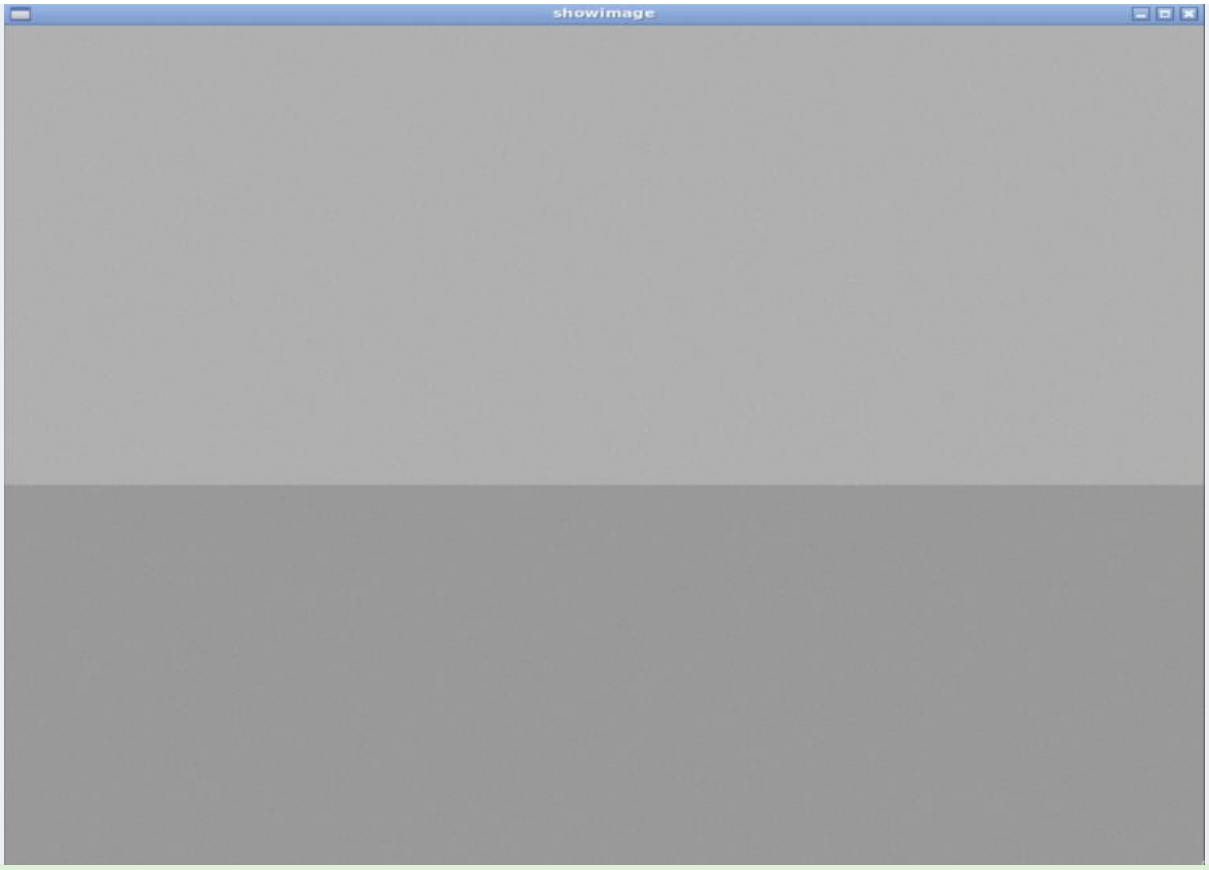
```
NO OUTPUT
```

Shell #3 Output: Cam2image

```
[INFO] [showimage]: Received image #camera_link
...
```

Now, you can open the **Graphical Tools** and you should see something like this:





- End of Example 3.1 -

- Example 3.2 -

Move Robot from ROS1-Gazebo with ROS2

Let's move the robot that is working with ROS1 and Gazebo, through ROS2 with the help of ROS1-Bridge.
For this, you will need **two** different webshells, which we will name (for future reference):

- WebShell 1: ROS1-Bridge
- WebShell 3: Publish movement command through ROS2

Execute in Shell #1: ROS1-Bridge

```
. /home/user/.bashrc_bridge
ros2 run ros1_bridge dynamic_bridge
```

Execute in Shell #2: Publish movement command through ROS2

To Make the robot turn:

```
. /home/user/.bashrc_ros2
ros2 topic pub /cmd_vel geometry_msgs/Twist "{linear:{x: 0.0,y: 0.0,z: 0.0}, angular:{x: 0.0,y: 0.0,z: 1.0}}"
```

Shell #1 Output: ROS1-Bridge

```
created 2to1 bridge for topic '/cmd_vel' with ROS 2 type 'geometry_msgs/msg/Twist' and ROS 1 type ''
[INFO] [1610047640.230618851] [ros_bridge]: Passing message from ROS 2 geometry_msgs/msg/Twist to ROS 1 geometry_msgs/Twist (showing msg only once per type)
[INFO] [1610047640.231388206] [ros_bridge]: Passing message from ROS 2 rcl_interfaces/msg/Log to ROS 1 rosgraph_msgs/Log (showing msg only once per type)
```

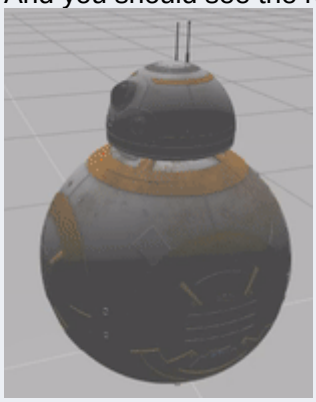
Shell #2 Output: Image topic remapper ROS1

```
publisher: beginning loop
```

publishing *#1: geometry_msgs.msg.Twist(linear=geometry_msgs.msg.Vector3(x=0.0, y=0.0, z=0.0), angular=geometry_msgs.msg.Vector3(x=0.0, y=0.0, z=1.0))*

...

And you should see the robot turning:



To Make the robot **STOP**:

```
. /home/user/.bashrc_ros2
ros2 topic pub /cmd_vel geometry_msgs/Twist "{linear:{x: 0.0,y: 0.0,z: 0.0}, angular:{x: 0.0,y: 0.0,z: 0.0}}"
```

- End of Example 3.2 -

- Example 3.3 -

Call the Reset Simulation service through ROS2

To practice service comunication with ROS2 using ROS1-Bridge, we are going to reset the simulation, calling its service in Gazebo, called **/gazebo/reset_simulation**
For this, you will need **two** different webshells, which we will name (for future reference):

- WebShell 1: ROS1-Bridge
- WebShell 2: Publish the service call with ROS2

Execute in Shell #1: ROS1-Bridge

```
. /home/user/.bashrc_bridge
ros2 run ros1_bridge dynamic_bridge
```

Execute in Shell #2: Call reset_simulation service through ROS2

To Make the robot turn:

```
. /home/user/.bashrc_ros2
ros2 service call /gazebo/reset_simulation std_srvs/Empty '{}'
```

Shell #1 Output: ROS1-Bridge

```
NO OUTPUT
```

Shell #2 Output

```
waiting for service to become available...
requester: making request: std_srvs.srv.Empty_Request()

response:
std_srvs.srv.Empty_Response()
```

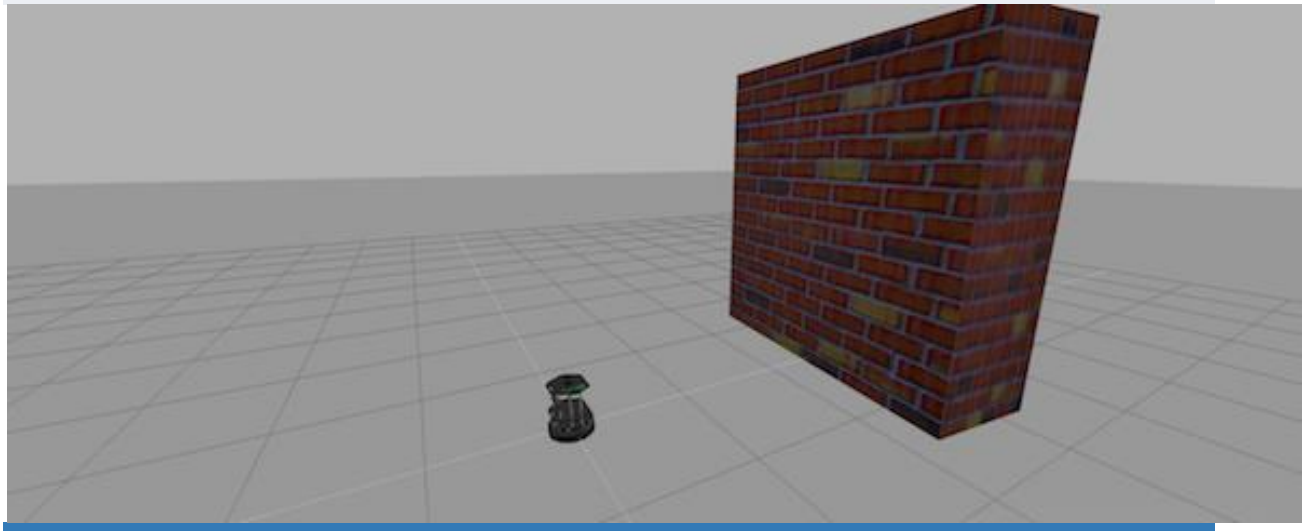
The robot should have been reset to its original state before you started tinkering with movement commands.

- End of Example 3.3 -

Congratulations, you can now combine ROS1 and ROS2.

ROS2 Basics in 5 days (C++)

Unit 4 ROS2 Topics: Publishers ¶



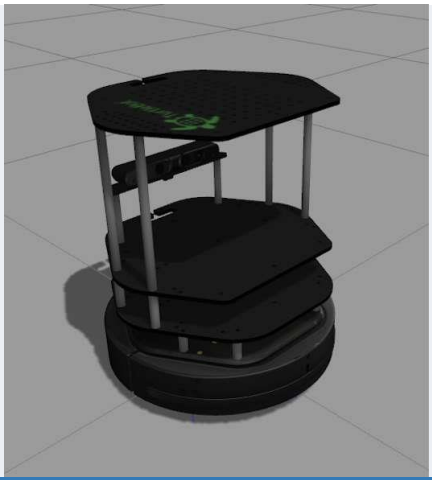
- Summary -

Estimated time to completion: 2.5 hours

Simulation: Turtlebot 2

What will you learn with this unit?

- What are topics and how to manage them
- What is a publisher and how to create one
- What are topic messages and how they work



- End of Summary -

4.1 Topic Publishers

- Exercise 4.1 -

- Create a new package named **topic_publisher_pkg**. When creating the package, add **rclcpp** and **std_msgs** as dependencies.
- Inside the src folder of the package, create a new file named **simple_topic_publisher.cpp**. Inside this file, copy the contents of simple_topic_publisher.cpp
- Create a launch file for launching this code.
- Do the necessary modifications to your **CMakeLists.txt** file, and compile the package.
- Execute the launch file to run your executable.

- End of Exercise 4.1 -

- Notes for Exercise 4.1 -

1.- Remember that in order to be able to use the ROS2 command line tools, you need to first source your environment properly with the following command:

```
source /opt/ros/foxy/setup.bash
```

2.- In order to do this exercise, you can simply follow the same steps you used in the previous chapter. It is almost the same.

3.- Remember, in order to create a package with **roscpp** and **std_msgs** as dependencies, you should use a command like the one below:

```
ros2 pkg create topic_publisher_pkg --build-type ament_cmake --dependencies rclcpp std_msgs
```

4.- The lines to add into the **CmakeLists.txt** file could be something like this:

```
add_executable(simple_publisher_node src/simple_topic_publisher.cpp)
ament_target_dependencies(simple_publisher_node rclcpp std_msgs)

install(TARGETS
  simple_publisher_node
  DESTINATION lib/${PROJECT_NAME}
 )

# Install launch files.
install(DIRECTORY
  launch
  DESTINATION share/${PROJECT_NAME}/
)
```

C++ Program {4.1}: simple_topic_publisher.cpp

```
#include "rclcpp/rclcpp.hpp"
#include "std_msgs/msg/int32.hpp"

int main(int argc, char * argv[])
{
  rclcpp::init(argc, argv);
  auto node = rclcpp::Node::make_shared("simple_publisher");
  auto publisher = node->create_publisher<std_msgs::msg::Int32>("counter", 10);
  auto message = std::make_shared<std_msgs::msg::Int32>();
  message->data = 0;
  rclcpp::WallRate loop_rate(2);

  while (rclcpp::ok()) {
```

```
    publisher->publish(*message);
    message->data++;
    rclcpp::spin_some(node);
    loop_rate.sleep();
  }
  rclcpp::shutdown();
  return 0;
}
```

END C++ Program {4.1}: simple_topic_publisher.cpp

- End of Notes -

Nothing happened? Well... that's not actually true! You have just created a topic named ***/counter***, and published through it as an integer that increases indefinitely. Let's check some things.

A topic is like a pipe. **Nodes use topics to publish information for other nodes** so that they can communicate.
You can find out, at any time, the number of topics in the system by doing a ***ros2 topic list***. You can also check for a specific topic.
On your webshell, type ***ros2 topic list*** and check for a topic named '/counter'.

Execute in Shell #2

```
ros2 topic list | grep  '/counter'
```

Shell #2 Output

```
user ~ $ ros2 topic list | grep '/counter'
/counter
```

Here, you have just listed all of the topics running right now and filtered with the **grep** command the ones that contain the word */counter*. If it appears, then the topic is running as it should.

You can request information about a topic by doing ***ros2 topic info <name_of_topic>***.
Now, type ***ros2 topic info /counter***.

Execute in Shell #2

```
ros2 topic info /counter
```

Shell #2 Output

```
user:~$ ros2 topic info /counter
Type: std_msgs/msg/Int32
Publisher count: 1
Subscription count: 0
```

The output indicates the name of the topic, and the number of Publishers/Subscribers that the topic has. At this moment, as you can see, it has only one Publisher, which is our program.
Now, type ***rostopic echo /counter*** and check the output of the topic in real-time.

Execute in Shell #2

```
ros2 topic echo /counter
```

You should see a succession of consecutive numbers, similar to the following:

Shell #2 Output

```
user:~$ ros2 topic echo /counter
data: 59

data: 60
```

```
data: 61

data: 62

data: 63

data: 64

data: 65

data: 66

data: 67
```

Ok, so... what has just happened? Let's explain it in more detail. First, let's crumble the code we've executed. You can check the comments in the code below, explaining what each line of the code does:

```
// Import all the necessary ROS libraries and import the Int32 message from the std_msgs package
#include "rclcpp/rclcpp.hpp"
#include "std_msgs/msg/int32.hpp"

using namespace std::chrono_literals;

int main(int argc, char * argv[])
{
  rclcpp::init(argc, argv);
  // Initiate a Node named 'simple_publisher'
  auto node = rclcpp::Node::make_shared("simple_publisher");
  // Create a Publisher object that will publish on the /counter topic, messages of the type Int32
  // The 10 indicates the size of the queue
  auto publisher = node->create_publisher<std_msgs::msg::Int32>("counter", 10);
  // Create a variable named 'message' of type Int32
  auto message = std::make_shared<std_msgs::msg::Int32>();
  // Initialize the 'message' variable
  message->data = 0;
  // Set a publish rate of 2 Hz
  rclcpp::WallRate loop_rate(2);

  // Create a loop that will go until someone stops the program execution
  while (rclcpp::ok()) {

    // Publish the message within the 'message' variable
    publisher->publish(*message);
    // Increment the 'message' variable
    message->data++;
    rclcpp::spin_some(node);
    // Make sure the publish rate maintains at 2 Hz
    loop_rate.sleep();
```

```
  }
  rclcpp::shutdown();
  return 0;
}
```

So basically, what this code does is *initiate a node and create a publisher that keeps publishing a sequence of consecutive integers into the '/counter' topic *. Summarizing:

A publisher is a node that keeps publishing a message into a topic. So now... what's a topic?

A topic is a channel that acts as a pipe, where other ROS nodes can either publish or read information. Let's now see some commands related to topics (some of them you've already used).
To **get a list of available topics** in a ROS system, you have to use the following command:

```
ros2 topic list
```

To **read the information that is being published in a topic,** use the following command:

```
ros2 topic echo <topic_name>
```

This command will start printing all of the information that is being published into the topic.
To **get information about a certain topic,** use the following command:

```
ros2 topic info <topic_name>
```

Finally, you can check the different options that *rostopic* command has by using the following command:

```
ros2 topic -h
```

4.2 Node Composition

In the previous example, you checked a C++ script called **simple_topic_publisher.cpp**. This script has been written using the **old school** programming method. We say this because it is very similar to the way you would have written this script in ROS1. In ROS2, though, this style of coding is going to become deprecated. And you may be asking... why? Well, it's because of **Composition**.

In ROS2, as a notable difference from ROS1, the concept of **Composition** is introduced. Basically, this means that you will have the ability to compose (execute) multiple nodes in a single process. You will learn more about **node composition** in following units.

In order to be able to use node composition, though, you will need to program your scripts in a more object-oriented way. So, the first script you checked, **simple_topic_publisher.cpp**, won't be able to use node composition.

Below, you can have a look at a script that does exactly the same thing, but it is coded using a composable method, using classes.

C++ Program {4.1b}: simple_topic_publisher_composable.cpp

```
#include "rclcpp/rclcpp.hpp"
#include "std_msgs/msg/int32.hpp"
#include <chrono>

using namespace std::chrono_literals;

/* This example creates a subclass of Node and uses std::bind() to register a
 * member function as a callback from the timer. */

class SimplePublisher : public rclcpp::Node
{
```

```
public:
  SimplePublisher()
  : Node("simple_publisher"), count_(0)
  {
    publisher_ = this->create_publisher<std_msgs::msg::Int32>("counter", 10);
    timer_ = this->create_wall_timer(
      500ms, std::bind(&SimplePublisher::timer_callback, this));
  }

private:
  void timer_callback()
  {
    auto message = std_msgs::msg::Int32();
    message.data = count_;
    count_++;
    publisher_->publish(message);
  }
  rclcpp::TimerBase::SharedPtr timer_;
  rclcpp::Publisher<std_msgs::msg::Int32>::SharedPtr publisher_;
  size_t count_;
};

int main(int argc, char * argv[])
{
  rclcpp::init(argc, argv);
  rclcpp::spin(std::make_shared<SimplePublisher>());
  rclcpp::shutdown();
  return 0;
}
```

END C++ Program {4.1b}: simple_topic_publisher_composable.cpp

4.2.1 Code Analysis

First, we define our class, which inherits from the **rclcpp::Node** class.

```
class SimplePublisher : public rclcpp::Node
```

Next, we have the constructor of our class:

```
SimplePublisher()
```

Within the constructor, we are initializing our node by calling to the constructor of the superclass **Node**, and also initializing a variable named **count**_ to 0.

```
: Node("simple_publisher"), count_(0)
```

Also within the constructor, we create our **publisher**_ and **timer**_ objects. As you will see later, they are actually created in the private section of our class, as **shared pointers** to these objects. Note that the timer object is bound to a function named **timer_callback**, which we will see next. This timer object will be triggered every 500ms.

```
publisher_ = this->create_publisher<std_msgs::msg::Int32>("counter", 10);
timer_ = this->create_wall_timer(
  500ms, std::bind(&SimplePublisher::timer_callback, this));
```

In the private section, we have the definition of the **timer_callback** function we introduced before. Inside this function, we are creating an **Int32 message**, which is given the value of the **count**_ variable. Then, we are going to increase the value of the count variable in 1, and we

are going to publish the message into our topic. Remember that this function will be called every 500ms, as defined in the timer_ object.

```
void timer_callback()
  {
    auto message = std_msgs::msg::Int32();
    message.data = count_;
    count_++;
    publisher_->publish(message);
  }
```

Also in the private section, we are creating the shared pointers to our publisher and timer objects defined above, and we are also creating the variable count.

```
rclcpp::TimerBase::SharedPtr timer_;
rclcpp::Publisher<std_msgs::msg::Int32>::SharedPtr publisher_;
size_t count_;
```

Finally, on the main function, all we do is create a SimplePublisher object, and make it spin until somebody terminates the program (Ctrl+C).

```
int main(int argc, char * argv[])
{
  rclcpp::init(argc, argv);
  rclcpp::spin(std::make_shared<SimplePublisher>());
  rclcpp::shutdown();
  return 0;
}
```

And that's it! Now, it is up to you to decide which method you will use for creating your ROS2 programs!

4.3 Messages (interfaces)

As you may have noticed, topics handle information through messages (interfaces). There are many different types of messages. In ROS1, we know this as messages. However, in ROS2 these messages are known as **interfaces**.

In the case of the code you executed before, the interface type was a **std_msgs/Int32**, but ROS2 provides a lot of different interfaces. You can even create your own interfaces, but it is recommended to use ROS default2 interfaces when possible.

Interfaces for topics are defined in **.msg** files, which are located inside a **msg** directory of a package.
To **get information about an interface,** use the following command:

```
ros2 interface show <message>
```

- Example 4.1 -

For example, let's try to get information about the std_msgs/Int32 message. Type the following command and check the output.

Execute in Shell #1

```
ros2 interface show std_msgs/msg/Int32
```

Shell #1 Output

```
# This was originally provided as an example message.
# It is deprecated as of Foxy
# It is recommended to create your own semantically meaningful message.
# However if you would like to continue using this please use the equivalent in example_msgs.
```

```
int32 data
```

In this case, the **Int32** interface has only one variable of type **int32**, named **data**. This Int32 interface comes from the package std_msgs, and you can find it in its **msg** directory.
Don't worry about the comments saying this message is deprecated, since this is just used as an example.

- End of Example 4.1 -

Now, you're ready to create your own publisher and make the robot move, so let's go for it!

- Exercise 4.2 -

- Modify the code you used previously so that it now publishes data to the /cmd_vel topic.
- Compile your package again.
- Launch the program and check that the robot moves.

- End of Exercise 4.2 -

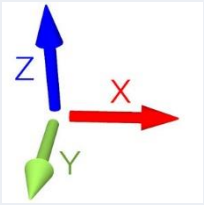
- Notes for Exercise 4.2 -

1.- The /cmd_vel topic is the topic used to move the robot.

2.- The type of message used by the /cmd_vel topic is **geometry_msgs/Twist**.

3.- In order to know the structure of the Twist messages, you need to use the **ros2 msg show** command.

4.- In this case, the robot uses a differential drive plugin to move. That is, the robot can only move linearly in the **x** axis, or rotationaly in the angular **z** axis. This means that the only values that you need to fill in the Twist message are the linear **x** and the angular **z**.



5.- The magnitudes of the Twist message are in m/s, so it is recommended to use values between 0 and 1. For example, 0.5 m/s.

6.- In order to be able to use the Twist message, you will need to include the geometry_msgs package in your **CMakeLists.txt** file. Here you can see how to do it:

```
find_package(geometry_msgs REQUIRED)

ament_target_dependencies(simple_publisher_node rclcpp std_msgs geometry_msgs)
```

7.- Remember that in order to be able to communicate with the Turtlebot2 robot, which is running on ROS1, you will need to start a ROS1 Bridge.

- End of Notes -

- Solutions for Exercise 4.2 -

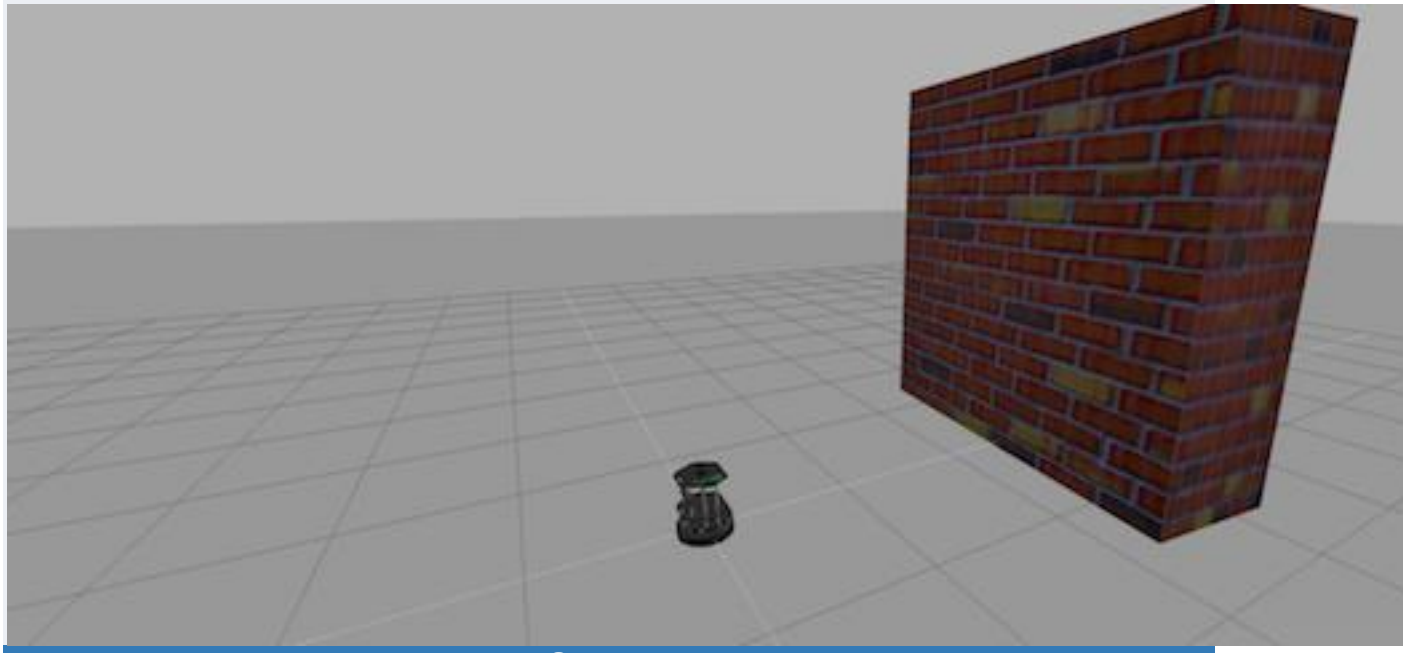
The Construct

Please try to do it by yourself unless you get stuck or need some inspiration. You will learn much more if you fight through each exercise.
Follow this link to open the solutions for the Topics Part 1: Topics Part 1 Solutions

- End of Solutions –

ROS2 Basics in 5 days (C++)

Unit 5 ROS2 Topics: Subscribers and interfaces ¶

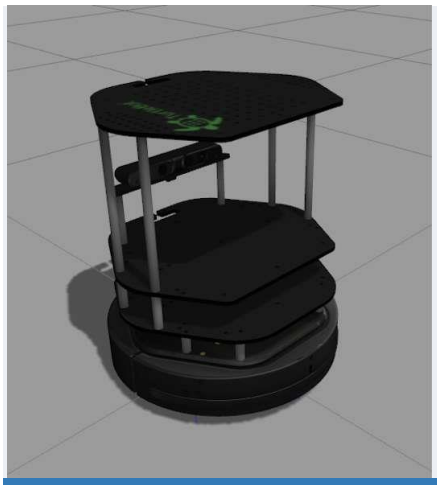


- Summary -

Estimated time to completion: 2.5 hours

What will you learn with this unit?

- What is a Subscriber and how to create one
- How to create your own interface



- End of Summary -

5.1 Topic Subscribers

You've learned that a topic is a channel where nodes can either write or read information. You've also seen that you can write into a topic using a publisher, so you may be thinking that there should also be some kind of similar tool to read information from a topic. And you're right! That's called a subscriber. **A subscriber is a node that reads information from a topic**. Let's execute the next code:

- Exercise 5.1 -

- Create a new package named **topic_subscriber_pkg**. When creating the package, add **rclcpp** and **std_msgs** as dependencies.
- Inside the src folder of the package, create a new file named **simple_topic_subscriber.cpp**. Inside this file, copy the contents of simple_topic_subscriber.cpp
- Create a launch file for launching this code.
- Do the necessary modifications to your **CMakeLists.txt** file, and compile the package.
- Execute the launch file to run your executable.

- End of Exercise 5.1 -

C++ Program {5.1}: simple_topic_subscriber.cpp

```
#include "rclcpp/rclcpp.hpp"
#include "std_msgs/msg/int32.hpp"
using std::placeholders::_1;

class SimpleSubscriber : public rclcpp::Node
{
public:
  SimpleSubscriber()
  : Node("simple_subscriber")
  {
    subscription_ = this->create_subscription<std_msgs::msg::Int32>(
      "counter", 10, std::bind(&SimpleSubscriber::topic_callback, this, _1));
  }

private:
```

```
  void topic_callback(const std_msgs::msg::Int32::SharedPtr msg)
  {
    RCLCPP_INFO(this->get_logger(), "I heard: '%d'", msg->data);
  }
  rclcpp::Subscription<std_msgs::msg::Int32>::SharedPtr subscription_;
};

int main(int argc, char * argv[])
{
  rclcpp::init(argc, argv);
  rclcpp::spin(std::make_shared<SimpleSubscriber>());
  rclcpp::shutdown();
  return 0;
}
```

END C++ Program {5.1}: simple_topic_subscriber.cpp

What's up? Nothing happened again? Well, that's not actually true... Let's do some checks.
Go to your webshell and type the following:

Execute in Shell #2

```
ros2 topic echo /counter
```

So... what happened? Nothing? And what does this mean? This means that **nobody is publishing into the /counter topic**, so there's no information to be read. Let's then publish something into the topic and see what happens. For that, let's introduce a new command:

```
ros2 topic pub <topic_name> <message_type> <value>
```

This command will publish the message you specify with the value you specify, in the topic you specify.
Open another webshell (leave the one with the ***rostopic echo*** open) and type the next command:

Execute in Shell #3

```
ros2 topic pub /counter std_msgs/Int32 "{data: '5'}"
```

Now, check the output of the console where you did the ***rostopic echo*** again. You should see something like this:

Shell #2 Output

```
user:~$ ros2 topic echo /counter
data: 5

data: 5

data: 5

data: 5

data: 5

...
```

This means that the value you published has been received by your subscriber program (which prints the value on the screen).
Now, check the output of the shell where you executed your subscriber code. You should see something like this:

```
ros2 launch topic_subscriber_pkg simple_topic_subscriber.launch.py
[INFO] [launch]: process[simple_subscriber_node-1]: started with pid [5900]
[INFO] [simple_subscriber]: I heard: '5'
[INFO] [simple_subscriber]: I heard: '5'
[INFO] [simple_subscriber]: I heard: '5'
[INFO] [simple_subscriber]: I heard: '5'
[INFO] [simple_subscriber]: I heard: '5'
[INFO] [simple_subscriber]: I heard: '5'
[INFO] [simple_subscriber]: I heard: '5'
```

Before explaining everything in more detail, let's explain the code you executed.

```
#include "rclcpp/rclcpp.hpp"
#include "std_msgs/msg/int32.hpp"
using std::placeholders::_1;

class SimpleSubscriber : public rclcpp::Node
{
public:
  // Initiate a Node called 'simple_subscriber'
  SimpleSubscriber()
  : Node("simple_subscriber")
  {
    // Create a Subscriber object that will listen to the /counter topic and will call the 'topic_callback' function
// each time it reads something from the topic
    subscription_ = this->create_subscription<std_msgs::msg::Int32>(
      "counter", 10, std::bind(&SimpleSubscriber::topic_callback, this, _1));
  }

private:
  // Define a function called 'topic_callback' that receives a parameter named 'msg'
  void topic_callback(const std_msgs::msg::Int32::SharedPtr msg)
  {
    // Print the value 'data' inside the 'msg' parameter
    RCLCPP_INFO(this->get_logger(), "I heard: '%d'", msg->data);
  }
  rclcpp::Subscription<std_msgs::msg::Int32>::SharedPtr subscription_;
};

int main(int argc, char * argv[])
{
  rclcpp::init(argc, argv);
  // Create a loop that will keep the program in execution
  rclcpp::spin(std::make_shared<SimpleSubscriber>());
  rclcpp::shutdown();
  return 0;
}
```

So, let's explain what has just happened. You've basically created a subscriber node that listens to the /counter topic, and each time it reads something, it calls a function that does a print of the msg. Initially, nothing happened since nobody was publishing into the /counter topic, but when

you executed the *ros2 topic pub* command, you published a message into the /counter topic, so your subscriber has printed that number and you could also see that message in the *ros2 topic echo* output. Now, everything makes sense, right?
Now, let's do some exercises to put into practice what you've learned!

- Exercise 5.2 -

Modify the previous code in order to print the odometry of the robot.

- End of Exercise 5.2 -

- Notes for Exercise 5.2 -

1. **The odometry of the robot is published by the robot into the */odom* topic.**
2. **You will need to figure out what message uses the */odom* topic, and the structure of this message.**
3. **Remember to compile your package again in order to update your executable.**

IMPORTANT NOTE: Remember that in order to be able to read the odometry data from the **/odom** topic of the simulation, you will need to first launch a ROS1 Bridge.

- End of Notes -

- Solution for Exercise 5.2 -

Please try to do it by yourself unless you get stuck or need some inspiration. You will learn much more if you fight through each exercise.
Follow this link to open the solutions notebook for Unit 5, Topics Part 2: Topics Part2 Solutions

- End of Solution -

- Exercise 5.3 -

1. Add to {Exercice 5.2} a C++ file that creates a publisher, which indicates the age of the robot, to the previous package.
2. For that, you'll need to create a new interface, called Age.msg. See the detailed description How to prepare CMakeLists.txt and package.xml for custom topic message compilation.

- End of Exercise 5.3 -

- Solution for Exercise 5.3 -

Please try to do it by yourself unless you get stuck or need some inspiration. You will learn much more if you fight through each exercise.
Follow this link to open the solutions notebook for Unit 5, Topics Part 2: Topics Part2 Solutions

- End of Solution -

5.2 How to create a custom interface

Now, you may be wondering... in case I need to publish some data that is not an Int32, which type of message should I use? You can use all ROS2 defined (***ros2 interface list***) messages. But, in case none fit your needs, you can create a new one.
In order to create a new message, you will need to do the following steps:

1. Create a directory named 'msg' inside your package
2. Inside this directory, create a file named Name_of_your_message.msg (more information below)
3. Modify CMakeLists.txt file (more information below)
4. Modify package.xml file (more information below)
5. Compile and source
6. Use in code

For example, let's create an interface that indicates age, with years, months, and days.
1) Create a directory msg in your package.

```
cd ~/ros2_ws/src/<package_name>
mkdir msg
```

2) The **Age.msg** file must contain this:

```
float32 years
float32 months
float32 days
```

3) **In CMakeLists.txt**
You will have to edit two functions inside CMakeLists.txt:

- find_package()
- rosidl_generate_interfaces()

I. find_package()

This is where all the packages required to COMPILE the messages of the topics, services, and actions go. In package.xml, you have to state them as **build_depend** and **exec_depend**.

```
find_package(ament_cmake REQUIRED)
find_package(rclcpp REQUIRED)
find_package(std_msgs REQUIRED)
find_package(rosidl_default_generators REQUIRED)
```

II. rosidl_generate_interfaces()

This function includes all of the messages of this package (in the msg folder) to be compiled. The function should look like this.

```
rosidl_generate_interfaces(${PROJECT_NAME}
  "msg/Age.msg"
)
```

Summarizing, this is the minimum expression of what is needed for the CMakaelist.txt to work:
Note: Keep in mind that the name of the package in the following example is **new_msg**, so in your case, the name of the package may be different.

```
cmake_minimum_required(VERSION 3.5)
project(new_msg)
```

```
# Default to C99
if(NOT CMAKE_C_STANDARD)
  set(CMAKE_C_STANDARD 99)
endif()

# Default to C++14
if(NOT CMAKE_CXX_STANDARD)
  set(CMAKE_CXX_STANDARD 14)
endif()

if(CMAKE_COMPILER_IS_GNUCXX OR CMAKE_CXX_COMPILER_ID MATCHES "Clang")
  add_compile_options(-Wall -Wextra -Wpedantic)
endif()

# find dependencies
find_package(ament_cmake REQUIRED)
find_package(rclcpp REQUIRED)
find_package(std_msgs REQUIRED)
find_package(rosidl_default_generators REQUIRED)

if(BUILD_TESTING)
  find_package(ament_lint_auto REQUIRED)
  # the following line skips the linter which checks for copyrights
  # remove the line when a copyright and license is present in all source files
  set(ament_cmake_copyright_FOUND TRUE)
  # the following line skips cpplint (only works in a git repo)
  # remove the line when this package is a git repo
  set(ament_cmake_cpplint_FOUND TRUE)
  ament_lint_auto_find_test_dependencies()
endif()

rosidl_generate_interfaces(${PROJECT_NAME}
  "msg/Age.msg"
)

ament_package()
```

4) **Modify package.xml**

Add the following lines to the package.xml file.

```
<build_depend>rosidl_default_generators</build_depend>

<exec_depend>rosidl_default_runtime</exec_depend>

<member_of_group>rosidl_interface_packages</member_of_group>
```

This is the minimum expression of the package.xml

Note: Keep in mind that the name of the package in the following example is **new_msg**, so in your case, the name of the package may be different.

```
<?xml version="1.0"?>
```

```
<?xml-model href="http://download.ros.org/schema/package_format2.xsd" schematypens="http://www.w3.org/2001/XMLSchema"?>
<package format="3">
  <name>new_msg</name>
  <version>0.0.0</version>
  <description>TODO: Package description</description>
  <maintainer email="ubuntu@todo.todo">ubuntu</maintainer>
  <license>TODO: License declaration</license>

  <buildtool_depend>ament_cmake</buildtool_depend>

  <depend>rclcpp</depend>
  <depend>std_msgs</depend>

  <build_depend>rosidl_default_generators</build_depend>
  <exec_depend>rosidl_default_runtime</exec_depend>
  <member_of_group>rosidl_interface_packages</member_of_group>

  <test_depend>ament_lint_auto</test_depend>
  <test_depend>ament_lint_common</test_depend>

  <export>
    <build_type>ament_cmake</build_type>
  </export>
</package>
```

5) Now, you have to compile the msgs. To do this, you have to type in a WebShell:

Execute in Shell #1

```
cd ~/ros2_ws
colcon build --packages-select new_msg
source install/setup.bash
```

VERY IMPORTANT: When you compile new messages, there is still an extra step before you can use the messages. You have to type in the Webshell, in the **ros2_ws**, the following command: ***source install/setup.bash***.

This executes this bash file that sets, among other things, the newly generated messages created through the colcon build.

If you don't do this, it might give you an import error, saying it doesn't find the message generated.

HINT 2: To verify that your message has been created successfully, type into your webshell: *ros2 interface show topic_subscriber_pkg/msg/Age*. If the structure of the Age message appears, it will mean that your message has been created successfully and it's ready to be used in your ROS programs.

Execute in Shell #1

```
ros2 interface show topic_subscriber_pkg/msg/Age
```

Shell #1 Output

```
float32 years
float32 months
```

```
float32 days
```

5.3 Use custom interfaces

You will have to add to your **CMakeLists.txt** the following extra lines (Pay special attention to the comments in the code) to compile and link your executable (in this example, it's called **publish_age.cpp**):

```
find_package(new_msg REQUIRED) # This is the package that contains the custom interface

add_executable(age_publisher_node src/publish_age.cpp)
ament_target_dependencies(age_publisher_node rclcpp std_msgs new_msg) # Note that we are also adding the package which contains the custom interface as a dependency of the node that will be using this custom interfac
e

install(TARGETS
  age_publisher_node
  DESTINATION lib/${PROJECT_NAME}
)
```

Finally, you will also need the add the package as a dependency in your **package.xml** file:

```
<depend>new_msg</depend>
```

5.4 Topics Mini Project

With all you've learned during this course, you're now able to do a small quiz to put everything together. Subscribers, Publishers, Messages... you will need to use all of these concepts in order to succeed!
In this project, you will create a code to make the robot avoid the wall that is in front of him. To help you achieve this, let's divide the project down into smaller units:

1. Create a Publisher that writes into the **/cmd_vel** topic in order to move the robot.
2. Create a Subscriber that reads from the **/kobuki/laser/scan topic**. This is the topic where the laser publishes its data.
3. Depending on the readings you receive from the laser's topic, you'll have to change the data you're sending to the /cmd_vel topic, in order to avoid the wall. This means, use the values of the laser to decide.

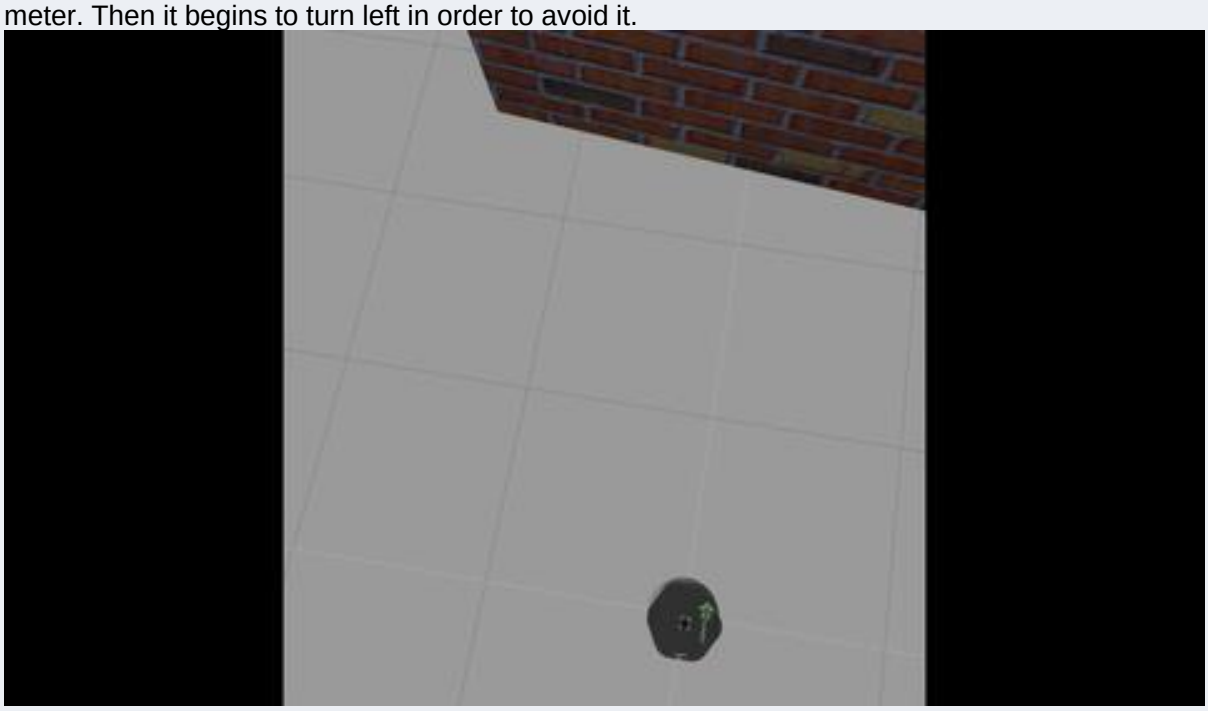
The logic that your program has to follow is the next one:

1. If the laser reading in front of the robot is higher than 1 meter (there is no obstacle closer than 1 meter in front of the robot), the robot will move forward.
2. If the laser reading in front of the robot is lower than 1 meter (there is an obstacle closer than 1 meter in front of the robot), the robot will turn left.
3. If the laser reading at the right side of the robot is lower than 1 meter (there is an obstacle closer than 1 meter at the right side of the robot), the robot will turn left.

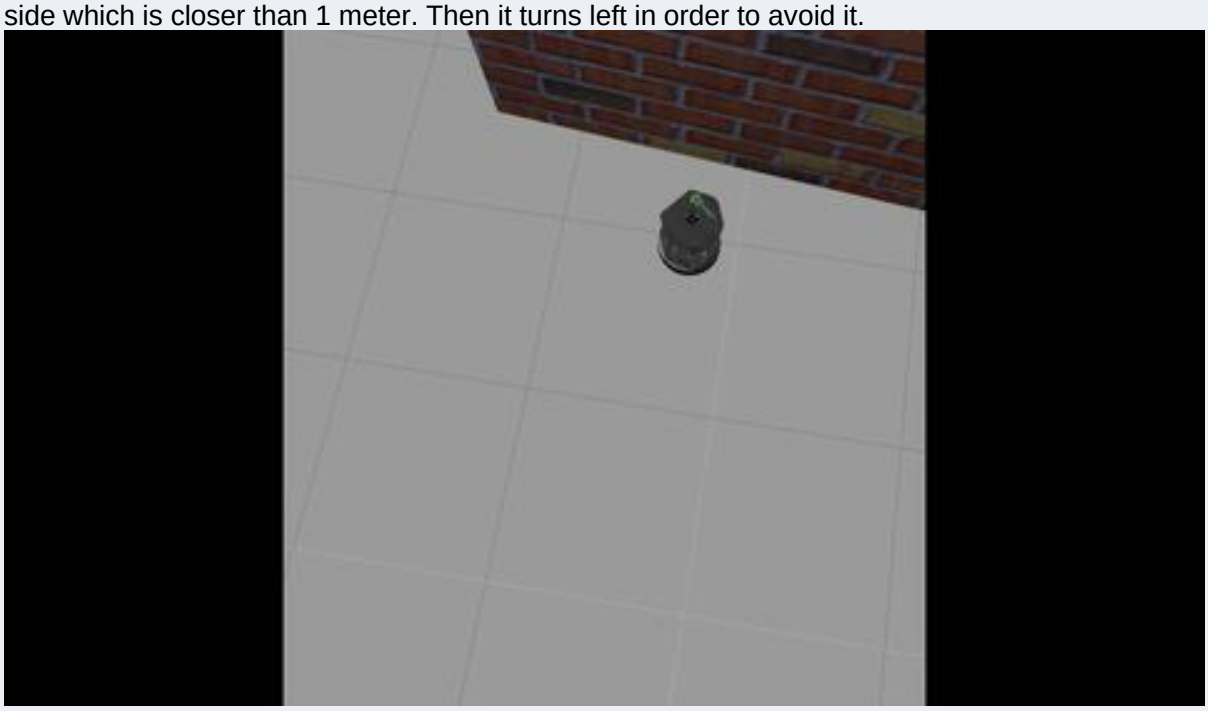
4. If the laser reading at the left side of the robot is lower than 1 meter (there is an obstacle closer than 1 meter at the left side of the robot), the robot will turn right.

The logic explained above has to result in a behavior like the following:

The robot starts moving forward until it detects an obstacle in front of it which is closer than 1 meter. Then it begins to turn left in order to avoid it.



The robot keeps turning left and moving forward until it detects that it has an obstacle at the right side which is closer than 1 meter. Then it turns left in order to avoid it.



Finally, the robot will continue moving forward since it won't detect any obstacle (closer than 1 meter) neither in front of it nor in its sides.
HINT 1: The data that is published into the /kobuki/laser/scan topic has a large structure. For this project, you just have to pay attention to the 'ranges' array.

Execute in Shell #1

```
ros2 interface show sensor_msgs/msg/LaserScan
```

Shell #1 Output

```
# Single scan from a planar laser range-finder
#
# If you have another ranging device with different behavior (e.g. a sonar
# array), please find or create a different message, since applications
# will make fairly laser-specific assumptions about this data

std_msgs/Header header # timestamp in the header is the acquisition time of
                      # the first ray in the scan.
                      #
                      # in frame frame_id, angles are measured around
                      # the positive Z axis (counterclockwise, if Z is up)
                      # with zero angle being forward along the x axis

float32 angle_min          # start angle of the scan [rad]
float32 angle_max          # end angle of the scan [rad]
float32 angle_increment    # angular distance between measurements [rad]

float32 time_increment     # time between measurements [seconds] - if your scanner
                      # is moving, this will be used in interpolating position
                      # of 3d points
float32 scan_time          # time between scans [seconds]

float32 range_min          # minimum range value [m]
float32 range_max          # maximum range value [m]

float32[] ranges           # range data [m]
                      # (Note: values < range_min or > range_max should be discarded)
float32[] intensities      # intensity data [device-specific units].  If your
                      # device does not provide intensities, please leave
                      # the array empty.
```

HINT 2: The 'ranges' array has a lot of values. The ones that are in the middle of the array represent the distances that the laser is detecting right in front of him. This means that the values in the middle of the array will be the ones that detect the wall. So, in order to avoid the wall, you just have to read these values.
HINT 3: The laser has a range of 30m. When you get readings of values around 30, it means that the laser isn't detecting anything. If you get a value that is under 30, this will mean that the laser is detecting some kind of obstacle in that direction (the wall).
HINT 4: The scope of the laser is about 180 degrees from right to left. This means that the values at the beginning and at the end of the 'ranges' array will be the ones related to the readings on the sides of the laser (left and right), while the values in the middle of the array will be the ones related to the front of the laser.

- Solution for Mini Project -

Please try to do it by yourself unless you get stuck or need some inspiration. You will learn much more if you fight through each exercise.
Follow this link to open the solutions notebook for Unit 2, Topics Part 2: Topics Part2 Solutions

- End of Solution -

It is now time that you start the project for this course!

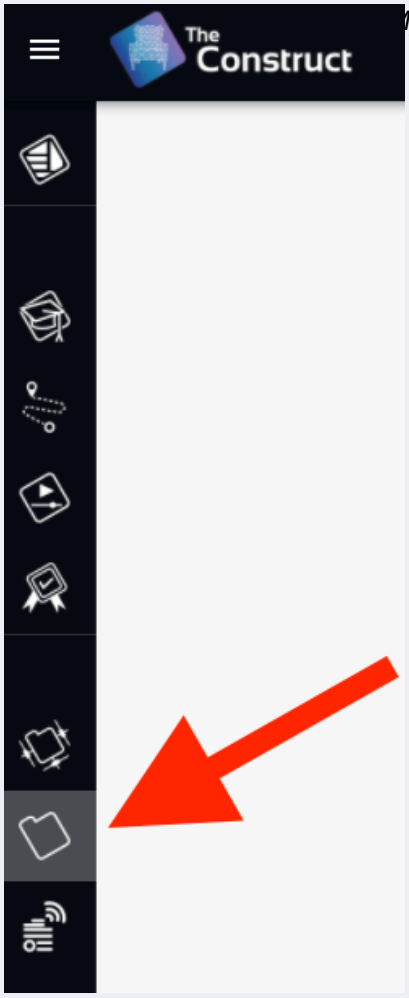
Each course in this academy includes a project that must be solved applying the knowledge gained from the whole course.

The project is going to be done in a different environment, which we call the **ROS Development Studio** (ROSDS). The ROSDS is an environment closer to what you will find when programming robots for companies. It is not as guided as this academy.

The ROSDS is included with your subscription, and is integrated inside The Construct. So no extra effort needs to be made by you. Well... you will need to expend some extra learning effort! But that's why you are here!

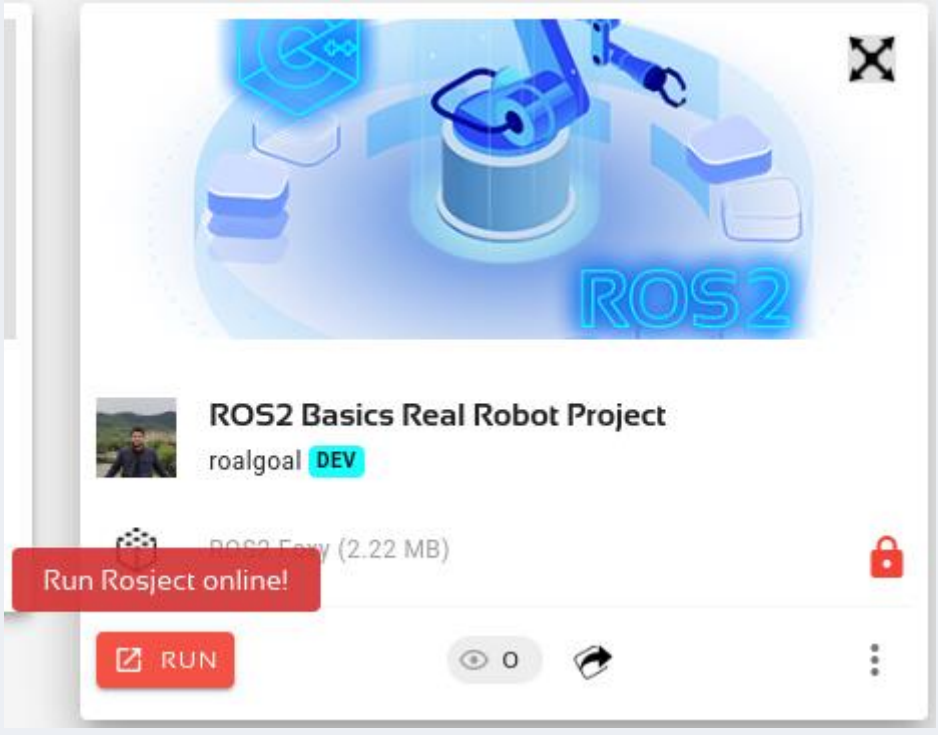
To start the project, you first need to get a copy of the ROS project (rosject), which contains the project instructions. Do the following:

1.- **Copy the project rosject** to your ROSDS area (see instructions below).



ly Rosjects area in The Construct

3.- **Open the rosject** by clicking *Run* on this course rosject



4.- Then follow the instructions of the rosject to **finish the PART I of the project**.

You can now copy the project rosject by clicking here. This will automatically make a copy of it.

You should finish PART I of the rosject before attempting next unit of this course!

Unit 6 Node Composition

As you have seen until now, the ROS2 programs (nodes) you create are compiled into executables that you can then execute, either directly or using a launch file. However, this method has a limitation: each executable will run in 1 process.

What does this mean? This means that **you can't run more than 1 ROS node in a single process**. Thankfully for us, ROS2 provides **components**. So... what are components?

6.1 What are ROS2 components?

Components are the equivalent in ROS2 to well-known ROS1 nodelets. By writing our program as a component, we can build it into a shared library instead of an executable. This allows us to load multiple components into a single process. Quite interesting, right?
However, in order to be able to use composition, you need to write your programs in a specific way, as you have already seen in previous units. In order to better understand how composition works, let's follow an example with the programs you created in unit 2.

6.2 Creating a component

- Notes -

This unit uses a ROS2 simulation. Therefore, it's not needed to run the ROS1 Bridge.

- End of Notes -

- Exercise 6.1 -

Let's start by creating a new package, where we will create our components:

Execute in Shell #1

```
$ source /opt/ros/foxy/setup.bash
$ cd ~/ros2_ws/src
$ ros2 pkg create my_components --build-type ament_cmake --dependencies rclcpp rclcpp_components compos
ition geometry_msgs
```

As you can see, we need to add dependencies to the **rclcpp_components** and **composition** packages.
First of all, you will have to create an **hpp** file to define your class. Inside the **include/my_components** folder of the package, create a new file named **moverobot_component.hpp** and paste the below code to it:

moverobot_component.hpp

```
#ifndef COMPOSITION__MOVEROBOT_COMPONENT_HPP_
#define COMPOSITION__MOVEROBOT_COMPONENT_HPP_

#include "my_components/visibility_control.h"
#include "rclcpp/rclcpp.hpp"
#include "geometry_msgs/msg/twist.hpp"

namespace my_components
{

class MoveRobot : public rclcpp::Node
{
public:
  COMPOSITION_PUBLIC
  explicit MoveRobot(const rclcpp::NodeOptions & options);

protected:
  void on_timer();

private:
  rclcpp::Publisher<geometry_msgs::msg::Twist>::SharedPtr pub_;
  rclcpp::TimerBase::SharedPtr timer_;
};

}  // namespace composition

#endif  // COMPOSITION__MOVEROBOT_COMPONENT_HPP_
```

As you can see, in this file we are just defining the different variables and functions that we will us in our **MoveRobot** class. Note also that we are encapsulating everything inside a **my_components** namespace.

Next we will create our program. Inside the **src** folder of the package, create a new script named **moverobot_component.cpp**, and paste the below code into it:

moverobot_component.cpp

```
#include "my_components/moverobot_component.hpp"

#include <chrono>
#include <iostream>
#include <memory>
#include <utility>

#include "rclcpp/rclcpp.hpp"
#include "std_msgs/msg/string.hpp"
#include "geometry_msgs/msg/twist.hpp"

using namespace std::chrono_literals;

namespace my_components
{
MoveRobot::MoveRobot(const rclcpp::NodeOptions & options)
: Node("moverobot", options)
{

  pub_ = create_publisher<geometry_msgs::msg::Twist>("cmd_vel", 10);

  timer_ = create_wall_timer(1s, std::bind(&MoveRobot::on_timer, this));
}

void MoveRobot::on_timer()
{
  auto msg = std::make_unique<geometry_msgs::msg::Twist>();
  msg->linear.x = 0.3;
  msg->angular.z = 0.3;
  std::flush(std::cout);

  pub_->publish(std::move(msg));
}

}

#include "rclcpp_components/register_node_macro.hpp"

RCLCPP_COMPONENTS_REGISTER_NODE(my_components::MoveRobot)
```

Let's point out the most important parts of the code.

First, we are importing the **.hpp** file we just created:

```
#include "my_components/moverobot_component.hpp"
```

Note that we are also encapsulating everything inside the **my_components** namespace:

```
namespace my_components
```

```
{
}
```

Finally, we register our program as a component:

```
#include "rclcpp_components/register_node_macro.hpp"

RCLCPP_COMPONENTS_REGISTER_NODE(my_components::MoveRobot)
```

The rest of the program is just the same as the one you created in unit 2.

- Notes -

Note that, since a component is only built into a shared library, it doesn't have a main function.

- End of Notes -

Let's now create another file, also inside the **include/my_components** folder, named **visibility_control.h**. Copy the code shown below.

visibility_control.h

```
// Copyright 2016 Open Source Robotics Foundation, Inc.
//
// Licensed under the Apache License, Version 2.0 (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at
//
//     http://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to in writing, software
// distributed under the License is distributed on an "AS IS" BASIS,
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
// See the License for the specific language governing permissions and
// limitations under the License.

#ifndef COMPOSITION__VISIBILITY_CONTROL_H_
#define COMPOSITION__VISIBILITY_CONTROL_H_

#ifdef __cplusplus
extern "C"
{
#endif

// This logic was borrowed (then namespaced) from the examples on the gcc wiki:
//     https://gcc.gnu.org/wiki/Visibility

#if defined _WIN32 || defined __CYGWIN__
  #ifdef __GNUC__
    #define COMPOSITION_EXPORT __attribute__ ((dllexport))
    #define COMPOSITION_IMPORT __attribute__ ((dllimport))
```

```
  #else
    #define COMPOSITION_EXPORT __declspec(dllexport)
    #define COMPOSITION_IMPORT __declspec(dllimport)
  #endif
  #ifdef COMPOSITION_BUILDING_DLL
    #define COMPOSITION_PUBLIC COMPOSITION_EXPORT
  #else
    #define COMPOSITION_PUBLIC COMPOSITION_IMPORT
  #endif
  #define COMPOSITION_PUBLIC_TYPE COMPOSITION_PUBLIC
  #define COMPOSITION_LOCAL
#else
  #define COMPOSITION_EXPORT __attribute__ ((visibility("default")))
  #define COMPOSITION_IMPORT
  #if __GNUC__ >= 4
    #define COMPOSITION_PUBLIC __attribute__ ((visibility("default")))
    #define COMPOSITION_LOCAL  __attribute__ ((visibility("hidden")))
  #else
    #define COMPOSITION_PUBLIC
    #define COMPOSITION_LOCAL
  #endif
  #define COMPOSITION_PUBLIC_TYPE
#endif

#ifdef __cplusplus
}
#endif

#endif  // COMPOSITION__VISIBILITY_CONTROL_H_
```

Basically, this file will optimize the loading process of shared libraries. If you'd like to get more details about this file, have a look at: https://gcc.gnu.org/wiki/Visibility

Time to compile!

Let's then modify the **CMakeLists.txt** file in order to generate the proper shared library from the component you have just created.

Add to CMakeLists.txt file

```
include_directories(include)

add_library(moverobot_component SHARED src/moverobot_component.cpp)
target_compile_definitions(moverobot_component PRIVATE "COMPOSITION_BUILDING_DLL")
ament_target_dependencies(moverobot_component
  "rclcpp"
  "rclcpp_components"
  "geometry_msgs")
rclcpp_components_register_nodes(moverobot_component "my_components::MoveRobot")
set(node_plugins "${node_plugins}my_components::MoveRobot;$<TARGET_FILE:moverobot_component>\n")

install(TARGETS
```

```
  moverobot_component
  ARCHIVE DESTINATION lib
  LIBRARY DESTINATION lib
  RUNTIME DESTINATION bin)
```

Let's point out the most important parts of the code.

First we generate the shared library from our C++ script:

```
add_library(moverobot_component SHARED src/moverobot_component.cpp)
```

We register our component:

```
rclcpp_components_register_nodes(moverobot_component "my_components::MoveRobot")
set(node_plugins "${node_plugins}my_components::MoveRobot;$<TARGET_FILE:moverobot_component>\n")
```

And finally we install it into our workspace:

```
install(TARGETS
  moverobot_component
  ARCHIVE DESTINATION lib
  LIBRARY DESTINATION lib
  RUNTIME DESTINATION bin)
```

Awesome! Let's then proceed to compile the package:

Execute in Shell #1

```
$ cd ~/ros2_ws
$ colcon build
$ source ~/ros2_ws/install/setup.bash
```

In order to check if the component has been created successfully, you can use the command ***ros2 component types***. This command will retur a list with all the available components:

Verify that the component has been created successfully:

Execute in Shell #1

```
$ ros2 component types
```

Amongst the list of components, you should see the component you just created:

Shell #1 Output

```
user:~/ros2_ws$ ros2 component types
my_components
  my_components::MoveRobot
composition
  composition::Talker
  composition::Listener
  composition::NodeLikeListener
  composition::Server
```

In order to be able to load a component, you have to first start the **component container**. Execute the following command:

Execute in Shell #1

```
$ ros2 run rclcpp_components component_container
```

Now, we can verify that the container is running with the following command:

Execute in Shell #2

```
$ ros2 component list
```

Shell #2 Output

```
user:~$ ros2 component list
/ComponentManager
```

Alright! Everything is set up, so let's load our component. Execute the following command:

Execute in Shell #2

```
$ ros2 component load /ComponentManager my_components my_components::MoveRobot
```

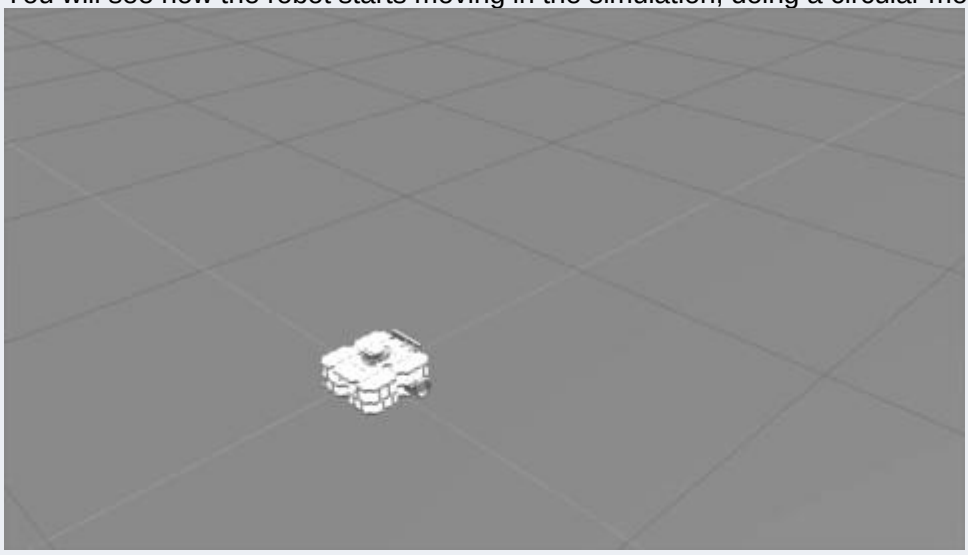
The structure of the command is the following:

```
ros2 component load /ComponentManager <pkg_name> <component_name>
```

Shell #2 Output

```
Loaded component 1 into '/ComponentManager' container node as '/moverobot'
```

You will see how the robot starts moving in the simulation, doing a circular movement:



In Shell #1, you will get the following output:

Shell #1 Output

```
[INFO] [1616686686.559639974] [ComponentManager]: Load Library: /home/user/ros2_ws/install/my_components/lib/libmoverobot_component.so
[INFO] [1616686686.560555430] [ComponentManager]: Found class: rclcpp_components::NodeFactoryTemplate<my_components::MoveRobot>
[INFO] [1616686686.560596502] [ComponentManager]: Instantiate class: rclcpp_components::NodeFactoryTemplate<my_components::MoveRobot>
```

As you can see, the process taken is the following:

- First, the component manager loads the program as a shared library.
- Second, it finds the **MoveRobot** class defined inside the component.
- Finally, it instantiates this **MoveRobot** class as a ROS node.

You can also make sure that your component has been loaded correctly using the ***ros2 component list*** command:

Execute in Shell #2

```
$ ros2 component list
```

Shell #2 Output

```
user:~/ros2_ws$ ros2 component list
/ComponentManager
  1   /moverobot
```

As you can see, the component **/moverobot** is identified with the number **1**. All components loaded will be identified with their own number.

And voilà! You have just created and loaded your first ROS2 component!
Of course, it's also possible to unload components. As you may imagine, it's done with the command ***ros2 component unload***:

Execute in Shell #2

```
$ ros2 component unload /ComponentManager 1
```

The structure of the command is the following:

```
ros2 component unload /ComponentManager <component_id>
```

Shell #2 Output

```
Unloaded component 1 from '/ComponentManager' container node
```

- Notes -

Note that, despite unloading the component, the robot will keep moving. To stop the robot, you can send a message to the **/cmd_vel** topic with:

```
$ ros2 topic pub --once /cmd_vel geometry_msgs/msg/Twist "{linear: {x:
0.0, y: 0.0, z: 0.0}, angular: {x: 0.0, y: 0.0, z: 0.0}}"
```

Remember that you can also reset the robot's initial position with the below command:

```
$ ros2 service call /reset_world std_srvs/Empty
```

- End of Notes -

It is also possible to load components using launch files. Inside your package, create a new folder named **launch**. Inside this folder, create a launch file named **moverobot_component.launch.py**, and paste the below code:

moverobot_component.launch.py

```
import launch
from launch_ros.actions import ComposableNodeContainer
from launch_ros.descriptions import ComposableNode


def generate_launch_description():
    """Generate launch description with multiple components."""
    container = ComposableNodeContainer(
            name='my_container',
            namespace='',
            package='rclcpp_components',
            executable='component_container',
            composable_node_descriptions=[
                ComposableNode(
                    package='my_components',
                    plugin='my_components::MoveRobot',
                    name='moverobot'),
            ],
            output='screen',
    )

    return launch.LaunchDescription([container])
```

Let's have a quick look at the launch file code. First, we are starting a component container, named **my_container**:

```
ComposableNodeContainer(
name='my_container',
namespace='',
package='rclcpp_components',
executable='component_container'
```

Second, we load the **my_components::MoveRobot** component into the container:

```
ComposableNode(
package='my_components',
plugin='my_components::MoveRobot',
name='moverobot')
```

- Notes -

As you have seen in the launch file code, we are starting a component container. Therefore, terminate the container you launched previously (probably running in Shell #1) in case you still have it running.

- End of Notes -

Don't forget to install the launch file in your worspace:

Add to CMakeLists.txt file

```
# Install launch files.
install(DIRECTORY
  launch
  DESTINATION share/${PROJECT_NAME}
)
```

Let's then proceed to compile the package:

Execute in Shell #1

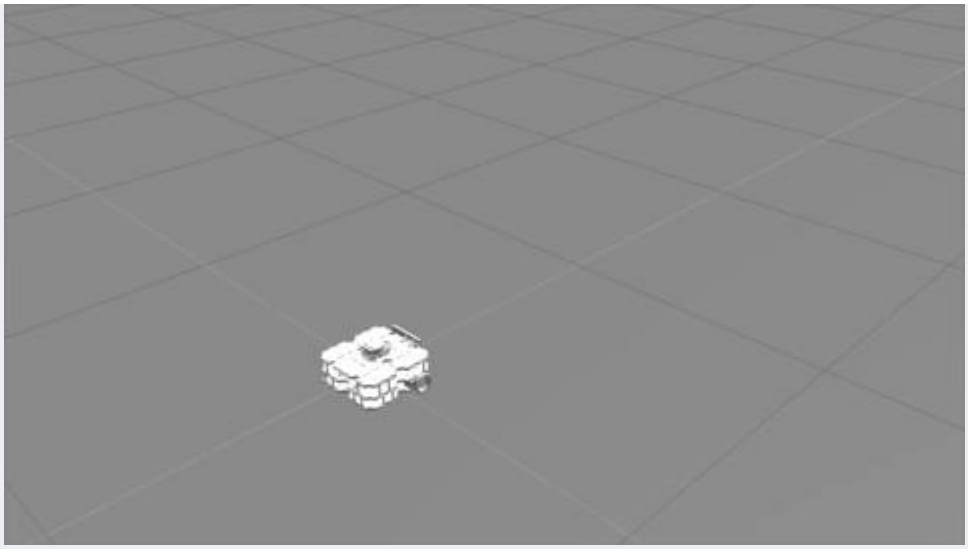
```
cd ~/ros2_ws
colcon build
source ~/ros2_ws/install/setup.bash
```

Now, let's execute our launch file:

Execute in Shell #1

```
$ ros2 launch my_components moverobot_component.launch.py
```

You will se the robot start moving again:



You can also execute the command ***ros2 component list*** to see what you get now:

Execute in Shell #2

```
ros2 component list
```

Shell #2 Output

```
user:~$ ros2 component list
/my_container
  1   /moverobot
```

- Notes -

When using a launch file to load components, it's not necessary to unload them using the command ***ros2 component unload***. When you stop the launch file execution (by pressing *Ctr+C*), all components will be automatically unloaded, alognside with the component container.

- End of Notes -

- End of Exercise 6.1 -

6.3 Summary

In the previous exercise you have seen many different commands that can be used to interact with ROS2 components. Let's do a brief recap of all of them:

- To see all available components:

```
ros2 component types
```

- To list all component that are currently running:

```
ros2 component list
```

- To start the component container:

```
ros2 run rclcpp_components component_container
```

- To load a component:

```
ros2 component load /ComponentManager <pkg_name> <component_name>
```

- To unload a component:

```
ros2 component unload /ComponentManager <component_id>
```

- Exercise 6.2 -

a) Create a new component, based on the program you created in exercise 2.4, that reads the odometry data of the robot.

b) Create a launch file that loads into a single process the 2 compononents created in this unit.

- End of Exercise 6.2 -

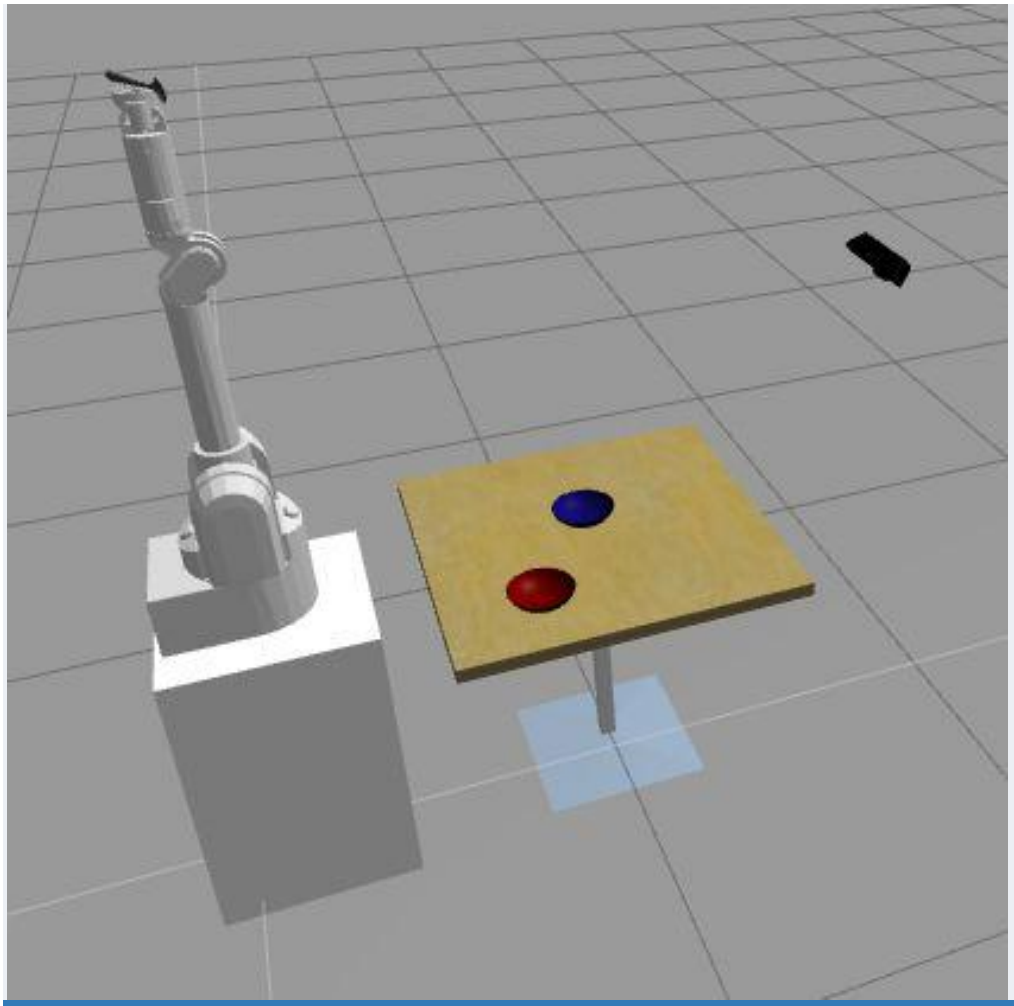
- Solutions for Exercise 6.2 -

Please try to do it by yourself unless you get stuck or need some inspiration. You will learn much more if you fight through each exercise.
Follow this link to open the solutions for Node Composition: Node Composition Solutions

- End of Solutions -

ROS2 Basics in 5 days (C++)

Unit 7 ROS2 Services: Clients



- Summary -

Estimated time to completion: 2.5 hours

What will you learn with this unit?

- What a service is
- How to create a service server
- How to create a service client
- How to call a service

- End of Summary -

Congratulations! You now know **75%** of ROS2 Basics!

With topics, you can do more or less whatever you want and need for your astromech droid. Many ROS2 packages only use topics and have the work perfectly done.

Then, why do you need to learn about **services**?

Well, that's because for some cases, topics are insufficient or just too cumbersome to use. Of course, you can destroy the *Death Star* with a stick, but you will just spend ages doing it. Better tell Luke SkyWalker to do it for you, right? Well, it's the same with services. They just make life easier.

7.1 Topics - Services - Actions

To understand what services are and when to use them, you have to compare them with topics and actions.

Imagine you have your own personal BB-8 robot. It has a laser sensor, a face-recognition system, and a navigation system. The laser will use a **Topic** to publish all of the laser readings at 20hz. We use a topic because we need to have that information available all the time for other ROS2 systems, such as the navigation system.

The face-recognition system will provide a **Service**. Your ROS2 program will call that service in order to get the name of the person BB-8 has in front of it.

The navigation system will provide an **Action**. Your ROS program will call the action to move the robot somewhere, and **WHILE** it's performing that task, your program will perform other tasks, such as complain about how tiring C-3PO is. And that action will give you **Feedback** (for example: distance left to the desired coordinates) as the robot moves toward the coordinates.
So... what's the difference between **Services** and **Topics**?

Services are based on a client-server (call-and-response) communication model. This means that **a service will ONLY be executed when it is specifically called**.

Topics, on the other hand, are based on a publish-subscribe model. This means that they will work with continuous flows of data.
And... what's the difference between a **Service** and an **Action**?

Well, to answer this question you'll have to wait until the next unit!

Conclusion: Use services when your program doesn't require to be executed contiuously.

ROS2 and Services Examples

We will see how to use commands for **ROS2 Service**.

We will also go step by step through the creation of a **Service Server** and **Service Client** for **ROS2**.

We will do the following:

- Go over the different commands for ROS2 services.
- Talk about limitations of ROS1-Bridge and how to circumvent them.
- Call through commands the Robot service to delete a model in ROS2.
- Create a dummy-Service-Server for ROS2.
- Create a client for the dummy_Service-Server for ROS2 to Delete a model.
- Modify the client for the dummy_Service-Server for ROS2, to call the real Robot service to delete a model.

7.2 Command Line Tools

ROS2, at the time of the creation of this tutorial, supports the following commands for services and its messages:

- **ros2 service list**: Lists all the services currently running in the system
- **ros2 service call**: Calls a certain service currently available
- **ros2 interface list**: Lists all the Service messages available
- **ros2 interface package**: Lists all the service messages from a package

- **ros2 interface packages**: Lists all the available packages that **have** service messages defined inside them.
- **ros2 interface show**: Gets the structure of a certain service message

WARNING: There is currently no support for the **ros2 service info** command that you would have in ROS1.

7.2.1 ros2 service list

- Notes -

In the next examples we will interact with a ROS1 service provided by Gazebo named **/gazebo/delete_model**. Therefore, because it's a **ROS1 Gazebo** service, we will need **ROS1-Bridge** to interact with it.

- End of Notes -

Remember that whenever you interact with ROS1, you need to launch **ROS1-Bridge**.

Execute in Shell #1

```
$ source .bashrc_bridge
$ ros2 run ros1_bridge dynamic_bridge
```

With this command, you can get all the services running now. In this case, most of them are related to ROS1-Gazebo.

Execute in Shell #2

```
# Get all the services currently running
$ source .bashrc_ros2
$ ros2 service list
```

Shell #2 Output

```
/camera/set_camera_info
/gazebo/clear_body_wrenches
/gazebo/clear_joint_forces
/gazebo/delete_light
/gazebo/delete_model
/gazebo/get_joint_properties
/gazebo/get_light_properties
/gazebo/get_link_properties
/gazebo/get_link_state
/gazebo/get_model_properties
/gazebo/get_model_state
/gazebo/get_physics_properties
/gazebo/get_world_properties
/gazebo/pause_physics
/gazebo/reset_simulation
/gazebo/reset_world
/gazebo/set_joint_properties
/gazebo/set_link_properties
/gazebo/set_link_state
/gazebo/set_model_configuration
/gazebo/set_model_state
/gazebo/set_physics_properties
/gazebo/spawn_sdf_model
/gazebo/spawn_urdf_model
```

/gazebo/unpause_physics

/ros_bridge/describe_parameters

/ros_bridge/get_parameter_types

/ros_bridge/get_parameters

/ros_bridge/list_parameters

/ros_bridge/set_parameters

/ros_bridge/set_parameters_atomically

7.2.2 ros2 service call

The command used in ROS2 to call service is ***ros2 service call***. The structure of this command is the following:

ros2 service call <service_name> <service_interface> <value>

Let's call the ***/gazebo/delete_model*** service:

Execute in Shell #2

```
# Call A Service Server
$ source .bashrc_ros2
$ ros2 service call /gazebo/delete_model gazebo_msgs/DeleteModel '{model_name: bowl_1}'
```

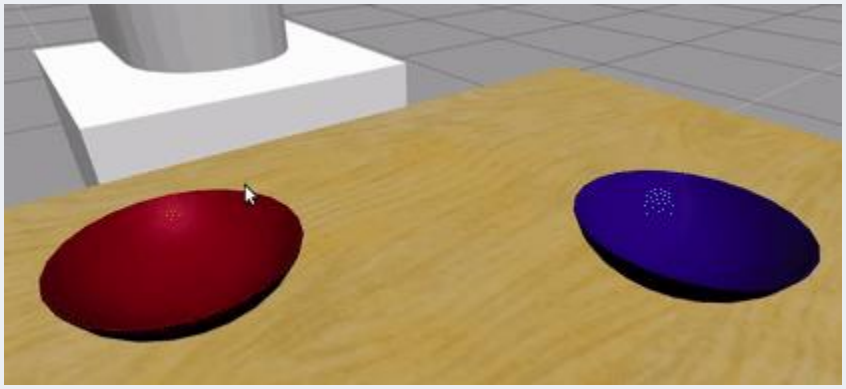
Note that you have to leave a space between ":" and the **name**. For further reference, please have a look at YAML Command Line from **ROS** documentation on how to fill in the calls for different messages. This is because in ROS2, at the time of the creation of this course, it doesn't support autocomplete the service messages in the call method in the command line.

Shell #2 Output

```
requester: making request: gazebo_msgs.srv.DeleteModel_Request(model_name='bowl_1')


response:
gazebo_msgs.srv.DeleteModel_Response(success=True, status_message='DeleteModel: successfully deleted m
odel')
```

You should now see in the simulation how the **bowl_1** gets deleted from the scene. If you want to reset the environment, just change to another unit that has a different simulation and come back to this one to reset the whole simulation environment.



7.2.3 ros2 interface list

Warning: Don't mix with the command **ros2 service list**. Here we are **NOT** listing the services currently available and running in the ROS2 system. Here we list the **Service interfaces** available for use in ROS2.

Execute in Shell #2

```
# List all available service messages in the system
$ source .bashrc_ros2
$ ros2 interface list | grep srv
```

Shell #2 Output

```
...
rcl_interfaces/srv/GetParameters
rcl_interfaces/srv/ListParameters
rcl_interfaces/srv/SetParameters
rcl_interfaces/srv/SetParametersAtomically
sensor_msgs/srv/SetCameraInfo
std_srvs/srv/Empty
std_srvs/srv/SetBool
std_srvs/srv/Trigger
tf2_msgs/srv/FrameGraph
...
```

As you can see, we are basically getting a list with all the interfaces and then filtering (using *grep*) only the results which are service messages (are inside a **srv** folder).

7.2.4 ros2 interface package

Execute in Shell #2

```
# List all the messages defined in a certain package
$ source .bashrc_ros2
$ ros2 interface package gazebo_msgs
```

Shell #2 Output

```
...
gazebo_msgs/GetModelState
gazebo_msgs/GetPhysicsProperties
gazebo_msgs/GetWorldProperties
gazebo_msgs/JointRequest
gazebo_msgs/SetJointProperties
gazebo_msgs/SetJointTrajectory
gazebo_msgs/SetLightProperties
gazebo_msgs/SetLinkProperties
gazebo_msgs/SetLinkState
...
```

Execute in Shell #2

```
# List all the messages defined in a certain package
$ source .bashrc_ros2
$ ros2 interface show gazebo_msgs/srv/DeleteModel
```

Shell #2 Output

```
# Deprecated, kept for ROS 1 bridge.
# Use DeleteEntity
string model_name               # name of the Gazebo Model to be deleted
---
bool success                    # return true if deletion is successful
string status_message           # comments if available
```

Note: As you can see in the comment, the **DeleteModel** message has been substituted in ROS2 by the **DelteEntity** message. However, we will use the old version within this example, since the simulation we are working with it's still based in ROS1.

7.3 Service interfaces

Does the structure of this **DeleteModel** message look familiar to you? It should, because it's the same structure as with topic interfaces, with some add-ons.

Service Message Properties:

- Service messages have the extension **.srv**. Remember that topic messages have the extension **.msg**
- Service messages are defined inside a **srv** directory, instead of a msg directory.
- Service messages have TWO parts:

```
**REQUEST**


---

**RESPONSE**
```

In the case of the DeleteModel service, **REQUEST** contains a string called ***model_name*** and **RESPONSE** is composed of a boolean named ***success***, and a string named ***status_message***.

The number of elements on each part of the service message can vary depending on the service needs. You can even put none if you find that it is irrelevant for your service. The important part of the message is the three dashes **---**, because they define the file as a Service Message. Summarizing:

The **REQUEST** is the part of the service message that defines **HOW you will do a call** to your service. This means, what variables you will have to pass to the Service Server so that it is able to complete its task.

The **RESPONSE** is the part of the service message that defines **HOW your service will respond** after completing its functionality. For instance, it will return a string with a certain message saying that everything went well, or it will return nothing, etc...

7.4 Create a Service Client

- Example 7.1 -

Create a ROS2 package

We first create the package where we will save all the service codes and exercises.

Execute in Shell #1

```
$ source .bashrc_ros2
$ cd ~/ros2_ws/src
$ ros2 pkg create cpp_unit_7_services --build-type ament_cmake --dependencies std_msgs rclcpp gazebo_msgs
```

Shell #1 Output

```
going to create a new package
package name: cpp_unit_7_services
destination directory: /home/user/ros2_ws/src
package format: 2
version: 0.0.0
description: TODO: Package description
maintainer: ['user <user@todo.todo>']
```

```
licenses: ['TODO: License declaration']
build type: ament_cmake
dependencies: ['std_msgs', 'rclcpp', 'gazebo_msgs']
creating folder ./cpp_unit_7_services
creating ./cpp_unit_7_services/package.xml
creating source and include folder
creating folder ./cpp_unit_7_services/src
creating folder ./cpp_unit_7_services/include/cpp_unit_7_services
creating ./cpp_unit_7_services/CMakeLists.txt
```

And now we compile.

Execute in Shell #1

```
$ cd ~/ros2_ws
# Compile all workspace
# colcon build --symlink-install
# Compile only the package we have created
$ colcon build --packages-select cpp_unit_7_services
```

Shell #1 Output

```
Starting >>> cpp_unit_7_services
Finished <<< cpp_unit_7_services [2.54s]

Summary: 1 package finished [2.73s]
```

Create the service client code

Let's now learn how to call a service in ROS2.

Execute in Shell #1

```
$ cd ~/ros2_ws/src/cpp_unit_7_services
$ touch src/cpp_simple_service_client.cpp
```

C++ Program {7.2}: cpp_simple_service_client.cpp

Note that if the original service message name was **DeleteModel.srv**, the generated **hpp** file for the messages will be named **delete_model.hpp**. If you had **MyCustomService.srv**, it would be **my_custom_service.hpp**, and so on.

```
#include <chrono>
#include <cinttypes>
#include <iostream>
#include <memory>
#include <string>

#include "rclcpp/rclcpp.hpp"
#include "gazebo_msgs/srv/delete_model.hpp"



gazebo_msgs::srv::DeleteModel::Response::SharedPtr send_request(
  rclcpp::Node::SharedPtr node,
  rclcpp::Client<gazebo_msgs::srv::DeleteModel>::SharedPtr client,
  gazebo_msgs::srv::DeleteModel::Request::SharedPtr request)
{

  auto result = client->async_send_request(request);
```

```
  // Wait for the result.
  if (rclcpp::spin_until_future_complete(node, result) ==
    rclcpp::executor::FutureReturnCode::SUCCESS)
  {

    RCLCPP_INFO(node->get_logger(), "Client request->model_name : %s", request->model_name.c_str());
    return result.get();
  } else {
    RCLCPP_ERROR(node->get_logger(), "service call failed :(");
    return NULL;
  }

}

int main(int argc, char ** argv)
{
    // Force flush of the stdout buffer.
    setvbuf(stdout, NULL, _IONBF, BUFSIZ);

    rclcpp::init(argc, argv);

    auto node = rclcpp::Node::make_shared("cpp_simple_service_client");
    auto topic = std::string("/gazebo/delete_model");
    auto client = node->create_client<gazebo_msgs::srv::DeleteModel>(topic);
    auto request = std::make_shared<gazebo_msgs::srv::DeleteModel::Request>();

    //  Fill the variable model_name of this object with the desired value
    request->model_name = "bowl_2";

    while (!client->wait_for_service(std::chrono::seconds(1))) {
    if (!rclcpp::ok()) {
      RCLCPP_ERROR(node->get_logger(), "Interrupted while waiting for the service. Exiting.");
      return 0;
    }
    RCLCPP_INFO(node->get_logger(), "service not available, waiting again...");
    }

    auto result = send_request(node, client, request);
    if (result) {

        auto result_str = result->success ? "True" : "False";

        RCLCPP_INFO(node->get_logger(), "Result-Success : %s", result_str);
        RCLCPP_INFO(node->get_logger(), "Result-Status: %s", result->status_message.c_str());
    } else {
        RCLCPP_ERROR(node->get_logger(), "Interrupted while waiting for response. Exiting.");
    }
```

```
    rclcpp::shutdown();
    return 0;
}
```

END C++ Program {7.2}: cpp_simple_service_client.cpp

Now, we make the necessary changes to the **CMakeLists.txt** file in order to compile the **client** code and install the executable in our workspace.

Setup {7.4}: CMakeLists.txt

```
cmake_minimum_required(VERSION 3.5)
project(cpp_unit_7_services)

# Default to C99
if(NOT CMAKE_C_STANDARD)
  set(CMAKE_C_STANDARD 99)
endif()

# Default to C++14
if(NOT CMAKE_CXX_STANDARD)
  set(CMAKE_CXX_STANDARD 14)
endif()

if(CMAKE_COMPILER_IS_GNUCXX OR CMAKE_CXX_COMPILER_ID MATCHES "Clang")
  add_compile_options(-Wall -Wextra -Wpedantic)
endif()

# find dependencies
find_package(ament_cmake REQUIRED)
find_package(std_msgs REQUIRED)
find_package(rclcpp REQUIRED)
find_package(gazebo_msgs REQUIRED)

if(BUILD_TESTING)
  find_package(ament_lint_auto REQUIRED)
  # the following line skips the linter which checks for copyrights
  # remove the line when a copyright and license is present in all source files
  set(ament_cmake_copyright_FOUND TRUE)
  # the following line skips cpplint (only works in a git repo)
  # remove the line when this package is a git repo
  set(ament_cmake_cpplint_FOUND TRUE)
  ament_lint_auto_find_test_dependencies()
endif()

add_executable(cpp_simple_service_client src/cpp_simple_service_client.cpp)
ament_target_dependencies(cpp_simple_service_client rclcpp std_msgs gazebo_msgs)

install(TARGETS
  cpp_simple_service_client
```

```
  DESTINATION lib/${PROJECT_NAME}
)


ament_package()
```

END Setup {7.4}: CMakeLists.txt

And now, we compile the whole **workspace**:

Execute in Shell #1

```
$ cd ~/ros2_ws
$ colcon build --packages-select cpp_unit_7_services
$ source install/setup.bash
```

Now, let's execute the **client** to check that it actually works:

Execute in Shell #2: ROS2 Service Client

```
$ ros2 run cpp_unit_7_services cpp_simple_service_client
```

Shell #2 Output

```
 [INFO] [1610107040.080234785] [cpp_simple_service_client]: Client request->model_name : bowl_2
[INFO] [1610107040.080474750] [cpp_simple_service_client]: Result-Success : True
[INFO] [1610107040.080514475] [cpp_simple_service_client]: Result-Status: DeleteModel: successfully delete
dmodel
```

- End of Example 7.1 -

- Exercise 7.1 -

Great, now it's time to test it by yourself with an exercise! In this exercise, you will have to create a client that removes the table from the simulation.

Execute in Shell #1

Remember that whenever you interact with ROS1, you need to launch **ROS1-Bridge**.

```
$ source .bashrc_bridge
$ ros2 run ros1_bridge dynamic_bridge
```

Execute in Shell #2

```
$ source .bashrc_ros2
$ ros2 run cpp_unit_7_services cpp_simple_service_client_ex7_1_node
```

- End of Exercise 7.1 -

- Solution for Exercise 7.1 -

Please try to do it by yourself unless you get stuck or need some inspiration. You will learn much more if you fight through each exercise.
Follow this link to open the solutions for the Services Part 1: Services Part 1 Solutions

- End of Solution -

Now, let's create a launch file to do the exact same thing you did in **Exercise 7.1**:

Execute in Shell #2

```
$ cd ~/ros2_ws/src/cpp_unit_7_services
$ mkdir launch
$ touch launch/start_cpp_simple_service_client_ex7_1.cpp.launch.py
$ chmod +x launch/start_cpp_simple_service_client_ex7_1.cpp.launch.py
```

C++ Program {7.5}: start_cpp_simple_service_client_ex7_1.cpp.launch.py

```
"""Launch cpp_simple_service_client_node_ex7_1"""

from launch import LaunchDescription
import launch_ros.actions


def generate_launch_description():
    return LaunchDescription([
        launch_ros.actions.Node(
            package='cpp_unit_7_services', executable='cpp_simple_service_client_ex7_1_node', output='screen'),
    ])
```

END C++ Program {7.5}: start_cpp_simple_service_client_ex7_1.cpp.launch.py

And add the following line to the **CMakelists.txt** file:

```
# Install launch files.
install(DIRECTORY
 launch
 DESTINATION share/${PROJECT_NAME}/
)
```

We now compile and execute:

Execute in Shell #2

```
$ cd ~/ros2_ws
$ colcon build --packages-select cpp_unit_7_services
$ source install/setup.bash
$ ros2 launch cpp_unit_7_services start_cpp_simple_service_client_ex7_1.cpp.launch.py
```

Shell #2 Output

```
 [INFO] [1610107040.080234785] [cpp_simple_service_client]: Client request->model_name : cafe_table
[INFO] [1610107040.080474750] [cpp_simple_service_client]: Result-Success : True
[INFO] [1610107040.080514475] [cpp_simple_service_client]: Result-Status: DeleteModel: successfully delete
dmodel
```

But don't get too excited deleting objects or you'll end up without a robot!

ROS2 Basics in 5 days (C++) ¶

Unit 8 ROS2 Services: Servers & interfaces ¶

- Summary -

Estimated time to completion: 3 hours

What will you learn with this unit?

- How to give a service
- How to create your own service server message

- End of Summary -

8.1 Service Server

In the previous lesson, you learned how to **CALL** a service server creating a service client. In this lesson, you are going to learn how to **CREATE** your own service server.

- Example 8.1 -

This is what we are going to do in this example:

- Inside the src folder of the package **cpp_unit_7_services**, which was created in a previous unit, create a new file named **empty_service_server.cpp**.

- Create a service server that uses the **std_srvs/Empty.srv** interface. When called, the service will print a LOG-INFO message. You can include this interface by adding the following line to your code:

```
#include "std_srvs/srv/empty.hpp"
```

- Create a launch file for launching this code.

- Do the necessary modifications to your **CMakeLists.txt** file, and compile the package.

- Execute the launch file to run the executable.

Execute in Shell #1

```
$ cd ~/ros2_ws/src/cpp_unit_7_services
$ touch src/empty_service_server.cpp
```

C++ Program: empty_service_server.cpp

```
#include <inttypes.h>
#include <memory>
#include "rclcpp/rclcpp.hpp"
#include "std_srvs/srv/empty.hpp"

using Empty = std_srvs::srv::Empty;
rclcpp::Node::SharedPtr g_node = nullptr;

void my_handle_service(
    const std::shared_ptr<rmw_request_id_t> request_header,
    const std::shared_ptr<Empty::Request> request,
    const std::shared_ptr<Empty::Response> response)
{
    (void)request_header;
    RCLCPP_INFO(g_node->get_logger(),"My_callback has been called");
}

int main(int argc, char ** argv)
{
    rclcpp::init(argc, argv);
    g_node = rclcpp::Node::make_shared("empty_service_server");
    auto server = g_node->create_service<Empty>("/my_service", my_handle_service);
    rclcpp::spin(g_node);
    rclcpp::shutdown();
    g_node = nullptr;
    return 0;
}
```

END C++ Program: empty_service_server.cpp

Now, we make the necessary changes to the **CMakeLists.txt** file to compile it. For that, just add these new lines:

Add to CMakeLists.txt

```
find_package(std_srvs REQUIRED)

add_executable(empty_service_server src/empty_service_server.cpp)
ament_target_dependencies(empty_service_server rclcpp std_msgs std_srvs)

install(TARGETS
  cpp_simple_service_client
  cpp_simple_service_client_ex7_1_node
  empty_service_server # Added
  DESTINATION lib/${PROJECT_NAME}
)
```

Note we've added some dependecies related to **std_srvs**

```
find_package(std_srvs REQUIRED)
...
ament_target_dependencies(empty_service_server rclcpp std_msgs std_srvs)
...
```

Your **CMakeLists.txt** file should look something like this:

Setup: CMakeLists.txt

```
cmake_minimum_required(VERSION 3.5)
project(cpp_unit_7_services)

# Default to C99
if(NOT CMAKE_C_STANDARD)
  set(CMAKE_C_STANDARD 99)
endif()

# Default to C++14
if(NOT CMAKE_CXX_STANDARD)
  set(CMAKE_CXX_STANDARD 14)
endif()

if(CMAKE_COMPILER_IS_GNUCXX OR CMAKE_CXX_COMPILER_ID MATCHES "Clang")
  add_compile_options(-Wall -Wextra -Wpedantic)
endif()

# find dependencies
find_package(ament_cmake REQUIRED)
find_package(std_msgs REQUIRED)
find_package(rclcpp REQUIRED)
find_package(gazebo_msgs REQUIRED)
find_package(std_srvs REQUIRED)

if(BUILD_TESTING)
  find_package(ament_lint_auto REQUIRED)
```

```
  # the following line skips the linter which checks for copyrights
  # uncomment the line when a copyright and license is not present in all source files
  #set(ament_cmake_copyright_FOUND TRUE)
  # the following line skips cpplint (only works in a git repo)
  # uncomment the line when this package is not in a git repo
  #set(ament_cmake_cpplint_FOUND TRUE)
  ament_lint_auto_find_test_dependencies()
endif()

add_executable(cpp_simple_service_client src/cpp_simple_service_client.cpp)
ament_target_dependencies(cpp_simple_service_client rclcpp std_msgs gazebo_msgs)

add_executable(cpp_simple_service_client_ex7_1_node src/cpp_simple_service_client_ex7_1.cpp)
ament_target_dependencies(cpp_simple_service_client_ex7_1_node rclcpp std_msgs gazebo_msgs)

add_executable(empty_service_server_node src/empty_service_server.cpp)
ament_target_dependencies(empty_service_server_node rclcpp std_msgs std_srvs)

install(TARGETS
   cpp_simple_service_client
   cpp_simple_service_client_ex7_1_node
   empty_service_server_node
   DESTINATION lib/${PROJECT_NAME}
 )

ament_package()
```

END Setup: CMakeLists.txt

Next we make the necessary changes to the **package.xml** file. In this case, just add this new line:

```
<depend>std_srvs</depend>
```

Your **package.xml** file should look something like this:

Setup: package.xml

```
<?xml version="1.0"?>
<?xml-model href="http://download.ros.org/schema/package_format2.xsd" schematypens="http://www.w3.org/2001/XMLSchema"?>
<package format="2">
  <name>cpp_unit_7_services</name>
  <version>0.0.0</version>
  <description>TODO: Package description</description>
  <maintainer email="user@todo.todo">user</maintainer>
  <license>TODO: License declaration</license>

  <buildtool_depend>ament_cmake</buildtool_depend>

  <depend>std_msgs</depend>
```

```
  <depend>rclcpp</depend>
  <depend>gazebo_msgs</depend>
  <depend>std_srvs</depend>


  <test_depend>ament_lint_auto</test_depend>
  <test_depend>ament_lint_common</test_depend>

  <export>
    <build_type>ament_cmake</build_type>
  </export>
</package>
```

END Setup: package.xml

And now, we compile the whole **cpp_unit_7_services** to update the changes:

Execute in Shell #1

```
$ source .bashrc_ros2
$ cd ~/ros2_ws
$ colcon build --packages-select cpp_unit_7_services
$ source install/setup.bash
```

Shell #1 Output

```
Starting >>> cpp_unit_7_services
--- stderr: cpp_unit_7_services
/home/user/ros2_ws/src/cpp_unit_7_services/src/empty_service_server.cpp: In function ‘void my_handle_service(std::shared_ptr<rmw_request_id_t>, std::shared_ptr<std_srvs::srv::Empty_Request_<std::allocator<void> > >, std::shared_ptr<std_srvs::srv::Empty_Response_<std::allocator<void> > >)’:
/home/user/ros2_ws/src/cpp_unit_7_services/src/empty_service_server.cpp:11:43: warning: unused parameter ‘request’ [-Wunused-parameter]
    const std::shared_ptr<Empty::Request> request,
                                          ^~~~~~~
/home/user/ros2_ws/src/cpp_unit_7_services/src/empty_service_server.cpp:12:44: warning: unused parameter ‘response’ [-Wunused-parameter]
    const std::shared_ptr<Empty::Response> response)
                                           ^~~~~~~~
---
Finished <<< cpp_unit_7_services [2.20s]

Summary: 1 package finished [2.32s]
  1 package had stderr output: cpp_unit_7_services
```

As you can see in the output, it warns you that the **request** and **response** variables are not used. This is normal because the **Empty.srv** doesn't really care about the call data and it doesn't respond in any way.

And let's finnaly execute your new **service server**:

Execute in Shell #1

```
$ source .bashrc_ros2
$ ros2 run cpp_unit_7_services empty_service_server_node
```

Did something happen?

Of course not! At the moment, you have just created and started the Service Server. So basically, you have made this service available for anyone to call it.
This means that if you run a ***ros2 service list***, you will be able to visualize this service among the list of available services.

Execute in Shell #2

```
$ source .bashrc_ros2
$ ros2 service list
```

Among the list of all available services, you should see the **/my_service** service.

```
/empty_service_server/describe_parameters
/empty_service_server/get_parameter_types
/empty_service_server/get_parameters
/empty_service_server/list_parameters
/empty_service_server/set_parameters
/empty_service_server/set_parameters_atomically
/my_service
```

Now, you have to actually **CALL** it.

So, let's call the **/my_service** service manually. Remember the calling structure discussed in the previous chapter.

Execute in Shell #2

```
$ ros2 service call /my_service std_srvs/Empty '{}'
```

Did it work? You should've seen the message, **'My callback function has been called'** printed at the output of the shell where you executed the service server code. And in the shell you executed the **call**, it should have given you some info that the call went well.

Shell #1 Output

```
[INFO] [empty_service_server]: My_callback has been called
```

Shell #2 Output

```
requester: making request: std_srvs.srv.Empty_Request()


response:
std_srvs.srv.Empty_Response()
```

We have to clear up that this is a very simple example. Normally, the request and response variables are used. An example is the **delete_model** service you used in the previous chapter.

For that case, you were passing the name of the object to delete to the Service Server in a variable called **model_name**. So. if you want to access the value of that **model_name** variable in the Service Server, you would have to do it like this:

```
request->model_name
```

Quite simple, right?
And to return the **RESPONSE** of the service, you have to access the variables in the **RESPONSE** part of the message. It would be like this:

```
response->success = true;
response->status_message = "The Model "+request->model_name+" was deleted.";
```

And why do we use **request** and **response** for accessing the **REQUEST** and **RESPONSE** parts of the service message? Well, it's because we are defining these variables here:

```
void handle_service(
    const std::shared_ptr<rmw_request_id_t> request_header,
    const std::shared_ptr<DeleteModel::Request> request,
```

```
const std::shared_ptr<DeleteModel::Response> response)
```

- End of Example 8.1 -

- Exercise 8.1 -

- The objective of this exercise is to create a service that, when called, makes BB-8 robot move in a square-like trajectory.

- You can work on a new package or use one of the ones you have already created.

- Create a C++ file that has a class inside. This class has to allow the movement of the BB-8 in a square-like movement {Fig-8.1}. This class could be called, for reference, **MoveBB8**. And the C++ file that contains it could be called **move_bb8.cpp**.

  To move the BB-8 robot, you just have to write into the **/cmd_vel** topic, as you did in the topics units.

- Notes -

Bear in mind that, although this is a simulation, BB-8 has weight and, therefore, it won't stop immediately due to inertia.

Also, when turning, friction and inertia will be playing a role. Remember that by only moving through */cmd_vel*, you don't have a way of checking if it turned the way you wanted it to (it's called an open loop system)... unless, of course, you find a way to have some positional feedback information. That's a challenge for advanced AstroMech builders (if you want to try, think about using the */odom* topic).

But for considering the movement, you just have to perform more or less a square movement. It doesn't have to be perfect.

- End of Notes -

- Add a Service Server that accepts an **Empty** Service message and activates the square movement. This service could be called **/move_bb8_in_square**.

  This activation will be done through a call to the Class that you have just generated, called **MoveBB8**.
  For that, you have to create a very similar C++ file as empty_service_server.cpp. You could call it **bb8_move_in_square_service_server.cpp**.

- Create a launch file called **start_bb8_move_in_square_service_server.launch.py**. Inside it, you have to start a node that launches the **bb8_move_in_square_service_server.cpp**.

- Launch **start_bb8_move_in_square_service_server.launch.py** and check that, when called through the Web Shell, BB-8 moves in a square.

- Create a new C++ file, called **bb8_move_in_square_service_client.cpp**, which calls the service **/move_bb8_in_square**. Remember how it was done in the previous chapter: **Services Part 1**.

  Then, generate a new launch file, called **call_bb8_move_in_square_service_server.launch.py**, which executes the code in the **bb8_move_in_square_service_client.cpp** file.

- Finally, when you launch this **call_bb8_move_in_square_service_server.launch** file, BB8 should move in a square.

- End of Exercise 8.1 -

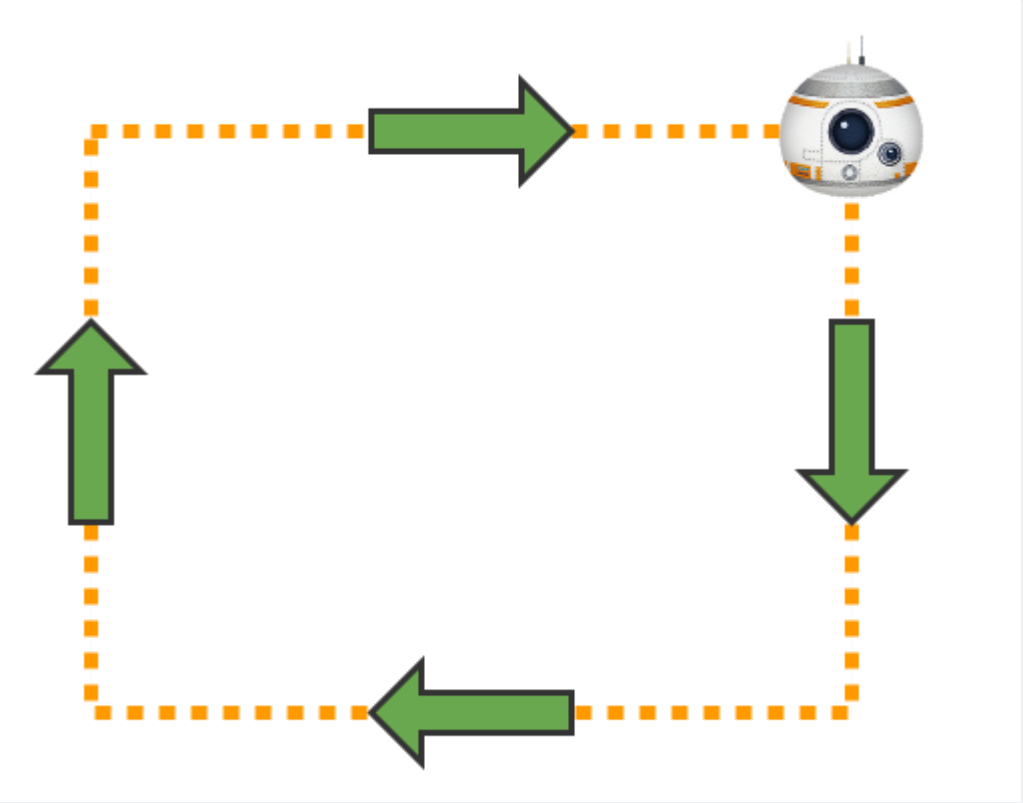


Fig.8.1 - BB8 Square Movement Diagram

- Solution for Exercise 8.1 -

Please try to do it by yourself unless you get stuck or need some inspiration. You will learn much more if you fight through each exercise.
Follow this link to open the solutions for the Services Part 2: Services Part 2 Solutions

- End of Solution -

8.2 How to create your own service interface

Create a New Package

Let's create a new package for the adding our custom service interface:

Execute in Shell #1

```
$ source .bashrc_ros2
$ cd ~/ros2_ws/src
$ ros2 pkg create unit_8_services_custom_msgs --dependencies std_msgs rclcpp
$ cd ~/ros2_ws
# We compile only our unit_8_services_custom_msgs package, nothing more
```

```
$ colcon build --packages-select unit_8_services_custom_msgs
```
Shell #1 Output

```
going to create a new package
package name: unit_8_services_custom_msgs
destination directory: /home/user/ros2_ws/src
package format: 2
version: 0.0.0
description: TODO: Package description
maintainer: ['user <user@todo.todo>']
licenses: ['TODO: License declaration']
build type: ament_cmake
dependencies: ['std_msgs', 'rclcpp']
creating folder ./unit_8_services_custom_msgs
creating ./unit_8_services_custom_msgs/package.xml
creating source and include folder
creating folder ./unit_8_services_custom_msgs/src
creating folder ./unit_8_services_custom_msgs/include/unit_8_services_custom_msgs
creating ./unit_8_services_custom_msgs/CMakeLists.txt
```

Create the custom service interface file

You can also create the ***MyCustomServiceMessage.srv*** through the IDE, if you don't feel confortable with vim.
The **MyCustomServiceMessage.srv** could be something like this:

Execute in Shell #1

```
$ cd ~/ros2_ws/src/unit_8_services_custom_msgs
$ mkdir srv
$ touch srv/MyCustomServiceMessage.srv
```

EXTRA: MyCustomServiceMessage.srv

```
float64 radius      # The distance of each side of the square
int32 repetitions   # The number of times BB-8 has to execute the square movement when the service is called
---
bool success        # Did it achieve it?
```

END EXTRA: MyCustomServiceMessage.srv

Prepare CMakeLists.txt and package.xml for Custom Service Compilation in ROS2

We would have to add changes to the following files:

- **CMakeLists.txt**: We have to add the necessary functions to generate the service interface wrappers. Also add dependencies that your custom interface needs.
- **package.xml**: Add the dependencies that your custom interface needs.

Setup: CMakeLists.txt

```
cmake_minimum_required(VERSION 3.5)
```

```
project(unit_8_services_custom_msgs)

# Default to C99
if(NOT CMAKE_C_STANDARD)
  set(CMAKE_C_STANDARD 99)
endif()

# Default to C++14
if(NOT CMAKE_CXX_STANDARD)
  set(CMAKE_CXX_STANDARD 14)
endif()

if(CMAKE_COMPILER_IS_GNUCXX OR CMAKE_CXX_COMPILER_ID MATCHES "Clang")
  add_compile_options(-Wall -Wextra -Wpedantic)
endif()

# find dependencies
find_package(ament_cmake REQUIRED)
find_package(std_msgs REQUIRED)
find_package(rclcpp REQUIRED)

# For Message Generation
find_package(rosidl_default_generators REQUIRED)

if(BUILD_TESTING)
  find_package(ament_lint_auto REQUIRED)
  # the following line skips the linter which checks for copyrights
  # remove the line when a copyright and license is present in all source files
  set(ament_cmake_copyright_FOUND TRUE)
  # the following line skips cpplint (only works in a git repo)
  # remove the line when this package is a git repo
  set(ament_cmake_cpplint_FOUND TRUE)
  ament_lint_auto_find_test_dependencies()
endif()

# For Message Generation
rosidl_generate_interfaces(${PROJECT_NAME}
  "srv/MyCustomServiceMessage.srv"
)

ament_package()
```

END Setup: CMakeLists.txt

Note that we are adding the package **rosidl_default_generators**, which is in charge of generating new interfaces in ROS2.

Also, note that we are calling the function **rosidl_generate_interfaces** in order to generate the new service interface.

```
find_package(rosidl_default_generators REQUIRED)

rosidl_generate_interfaces(${PROJECT_NAME}
  "srv/MyCustomServiceMessage.srv"
)
```

Setup: package.xml

```
<?xml version="1.0"?>
<?xml-model href="http://download.ros.org/schema/package_format2.xsd" schematypens="http://www.w3.org/2001/XMLSchema"?>
  <package format="3">
  <name>unit_8_services_custom_msgs</name>
  <version>0.0.0</version>
  <description>TODO: Package description</description>
  <maintainer email="ubuntu@todo.todo">ubuntu</maintainer>
  <license>TODO: License declaration</license>

  <buildtool_depend>ament_cmake</buildtool_depend>

  <depend>std_msgs</depend>
  <depend>rclcpp</depend>

  <build_depend>rosidl_default_generators</build_depend>
  <exec_depend>rosidl_default_runtime</exec_depend>
  <member_of_group>rosidl_interface_packages</member_of_group>

  <test_depend>ament_lint_auto</test_depend>
  <test_depend>ament_lint_common</test_depend>

  <export>
    <build_type>ament_cmake</build_type>
  </export>
</package>
```

END Setup: package.xml

We needed to add the following dependencies:

```
<build_depend>rosidl_default_generators</build_depend>
<exec_depend>rosidl_default_runtime</exec_depend>
<member_of_group>rosidl_interface_packages</member_of_group>
```

We add this **member_of_group** to avoid this error:

```
CMake Error at /opt/ros/bouncy/share/rosidl_cmake/cmake/rosidl_generate_interfaces.cmake:129 (message):
  Packages installing interfaces must include
  '<member_of_group>rosidl_interface_packages</member_of_group>' in their
  package.xml
```

Compile and generate the Custom interface

Execute in Shell #1

```
$ cd ~/ros2_ws
# We compile only our unit_8_services_custom_msgs package, nothing more
$ colcon build --packages-select unit_8_services_custom_msgs
$ source install/setup.bash
```

Shell #1 Output

```
Starting >>> unit_8_services_custom_msgs
[Processing: unit_8_services_custom_msgs]
Finished <<< unit_8_services_custom_msgs [31.0s]
```

We check that the messages were generated:

Execute in Shell #1

VERY IMPORTANT: After the message generation, you have to **SOURCE AGAIN**. Otherwise, you won't be able to see the messages generated through the ROS2 service commands, and you will think that it didn't work.

```
# Lista ll the service messages available and filter by name
$ ros2 interface list | grep unit_8_services_custom_msgs/srv/MyCustomServiceMessage
# Show contents of message
$ ros2 interface show unit_8_services_custom_msgs/srv/MyCustomServiceMessage
```

Shell #1 Output

```
float64 radius      # The distance of each side of the square
int32 repetitions   # The number of times BB-8 has to execute the square movement when the service is called
---
bool success        # Did it achieve it?
```

If you had this output, the message was generated correctly.
Great! So, to use it, you just have to add the following include in your cpp file:

```
#include "unit_8_services_custom_msgs/srv/my_custom_service_message.hpp"
```

Create a service server that uses this custom interface: MyCustomServiceMessage

To test that everything works, we will create a new service server that uses this custom message, and then call it through a python service client.

Execute in Shell #1

```
$ cd ~/ros2_ws/src/cpp_unit_7_services
$ touch src/custom_service_server.cpp
```

C++ Program: custom_service_server.cpp

```
#include <inttypes.h>
#include <memory>
#include "rclcpp/rclcpp.hpp"
#include "unit_8_services_custom_msgs/srv/my_custom_service_message.hpp"

using MyCustomServiceMessage = unit_8_services_custom_msgs::srv::MyCustomServiceMessage;
rclcpp::Node::SharedPtr g_node = nullptr;

void handle_service(
    const std::shared_ptr<rmw_request_id_t> request_header,
    const std::shared_ptr<MyCustomServiceMessage::Request> request,
    const std::shared_ptr<MyCustomServiceMessage::Response> response)
{
    (void)request_header;
```

```
    RCLCPP_INFO(g_node->get_logger(),"Incoming request\nradius: %f", request->radius);
    RCLCPP_INFO(g_node->get_logger(),"Incoming request\nrepetitions: %i", request->repetitions);

    response->success = true;
}

int main(int argc, char ** argv)
{
    rclcpp::init(argc, argv);
    g_node = rclcpp::Node::make_shared("custom_service_server");
    auto server = g_node->create_service<MyCustomServiceMessage>("/my_custom_service", handle_service);
    RCLCPP_INFO(g_node->get_logger(),"CustomServiceServer...READY");
    rclcpp::spin(g_node);
    rclcpp::shutdown();
    g_node = nullptr;
    return 0;
}
**END C++ Program: custom_service_server.cpp**
```

Execute in Shell #1

```
$ cd ~/ros2_ws/src/cpp_unit_7_services
$ touch src/custom_service_client.cpp
**C++ Program: custom_service_client.cpp**
#include <chrono>
#include <cinttypes>
#include <iostream>
#include <memory>
#include <string>

#include "rclcpp/rclcpp.hpp"
#include "unit_8_services_custom_msgs/srv/my_custom_service_message.hpp"

using MyCustomServiceMessage = unit_8_services_custom_msgs::srv::MyCustomServiceMessage;

MyCustomServiceMessage::Response::SharedPtr send_request(
  rclcpp::Node::SharedPtr node,
  rclcpp::Client<MyCustomServiceMessage>::SharedPtr client,
  MyCustomServiceMessage::Request::SharedPtr request)
{

  auto result = client->async_send_request(request);
  // Wait for the result.
  if (rclcpp::spin_until_future_complete(node, result) ==
    rclcpp::executor::FutureReturnCode::SUCCESS)
  {
    RCLCPP_INFO(node->get_logger(),"Client request->radius: %f", request->radius);
    RCLCPP_INFO(node->get_logger(),"Client request->repetitions: %i", request->repetitions);
```

```
    return result.get();
  } else {
    RCLCPP_ERROR(node->get_logger(), "service call failed :(");
    return NULL;
  }

}

int main(int argc, char ** argv)
{
    // Force flush of the stdout buffer.
    setvbuf(stdout, NULL, _IONBF, BUFSIZ);

    rclcpp::init(argc, argv);

    auto node = rclcpp::Node::make_shared("custom_service_client");
    auto topic = std::string("/my_custom_service");
    auto client = node->create_client<MyCustomServiceMessage>(topic);
    auto request = std::make_shared<MyCustomServiceMessage::Request>();

    //  Fill In The variables of the Custom Service Message
    request->radius = 2.3;
    request->repetitions = 2;

    while (!client->wait_for_service(std::chrono::seconds(1))) {
    if (!rclcpp::ok()) {
      RCLCPP_ERROR(node->get_logger(), "Interrupted while waiting for the service. Exiting.");
      return 0;
    }
    RCLCPP_INFO(node->get_logger(), "service not available, waiting again...");
    }

    auto result = send_request(node, client, request);
    if (result) {

        auto result_str = result->success ? "True" : "False";

        RCLCPP_INFO(node->get_logger(), "Result-Success : %s", result_str);
    } else {
        RCLCPP_ERROR(node->get_logger(), "Interrupted while waiting for response. Exiting.");
    }

    rclcpp::shutdown();
    return 0;
}
```

END C++ Program: custom_service_client.cpp

Now, we make the necessary changes to the **CMakeLists.txt** file to compile it.

```
find_package(unit_8_services_custom_msgs REQUIRED) # We add the package with the custom service interface as a dependency

...

# We generate the new server and client executables. Note that we are also adding the unit_8_services_custom_msgs as a dependency

add_executable(custom_service_server_node src/custom_service_server.cpp)
ament_target_dependencies(custom_service_server_node rclcpp std_msgs std_srvs unit_8_services_custom_msgs)

add_executable(custom_service_client_node src/custom_service_client.cpp)
ament_target_dependencies(custom_service_client_node rclcpp std_msgs std_srvs unit_8_services_custom_msgs)

...

install(TARGETS
  cpp_simple_service_client
  cpp_simple_service_client_ex7_1_node
  empty_service_server_node
  custom_service_server_node # We install the new executable
  custom_service_client_node # We install the new executable
  DESTINATION lib/${PROJECT_NAME}
)
```

Your **CMakeLists.txt** should look something like this:

Setup: CMakeLists.txt

```
cmake_minimum_required(VERSION 3.5)
project(cpp_unit_7_services)

# Default to C99
if(NOT CMAKE_C_STANDARD)
  set(CMAKE_C_STANDARD 99)
endif()

# Default to C++14
if(NOT CMAKE_CXX_STANDARD)
  set(CMAKE_CXX_STANDARD 14)
endif()

if(CMAKE_COMPILER_IS_GNUCXX OR CMAKE_CXX_COMPILER_ID MATCHES "Clang")
  add_compile_options(-Wall -Wextra -Wpedantic)
endif()

# find dependencies
find_package(ament_cmake REQUIRED)
```

```
find_package(std_msgs REQUIRED)
find_package(rclcpp REQUIRED)
find_package(gazebo_msgs REQUIRED)
find_package(std_srvs REQUIRED)
find_package(unit_8_services_custom_msgs REQUIRED)

if(BUILD_TESTING)
  find_package(ament_lint_auto REQUIRED)
  # the following line skips the linter which checks for copyrights
  # uncomment the line when a copyright and license is not present in all source files
  #set(ament_cmake_copyright_FOUND TRUE)
  # the following line skips cpplint (only works in a git repo)
  # uncomment the line when this package is not in a git repo
  #set(ament_cmake_cpplint_FOUND TRUE)
  ament_lint_auto_find_test_dependencies()
endif()

add_executable(cpp_simple_service_client src/cpp_simple_service_client.cpp)
ament_target_dependencies(cpp_simple_service_client rclcpp std_msgs gazebo_msgs)

add_executable(cpp_simple_service_client_ex7_1_node src/cpp_simple_service_client_ex7_1.cpp)
ament_target_dependencies(cpp_simple_service_client_ex7_1_node rclcpp std_msgs gazebo_msgs)

add_executable(empty_service_server_node src/empty_service_server.cpp)
ament_target_dependencies(empty_service_server_node rclcpp std_msgs std_srvs)

add_executable(custom_service_server_node src/custom_service_server.cpp)
ament_target_dependencies(custom_service_server_node rclcpp std_msgs std_srvs unit_8_services_custom_ms
gs)

add_executable(custom_service_client_node src/custom_service_client.cpp)
ament_target_dependencies(custom_service_client_node rclcpp std_msgs std_srvs unit_8_services_custom_ms
gs)

install(TARGETS
  cpp_simple_service_client
  cpp_simple_service_client_ex7_1_node
  empty_service_server_node
  custom_service_server_node
  custom_service_client_node
  DESTINATION lib/${PROJECT_NAME}
 )

ament_package()
```

END Setup: CMakeLists.txt

Now, we make the necessary changes to the **package.xml** file to compile it. For that, just add this new line:

```
<depend>unit_8_services_custom_msgs</depend>
```

Your **package.xml** should look something like this:

Setup: package.xml

```
<?xml version="1.0"?>
<?xml-model href="http://download.ros.org/schema/package_format2.xsd" schematypens="http://www.w3.org/2001/XMLSchema"?>
<package format="3">
  <name>cpp_unit_7_services</name>
  <version>0.0.0</version>
  <description>TODO: Package description</description>
  <maintainer email="user@todo.todo">user</maintainer>
  <license>TODO: License declaration</license>

  <buildtool_depend>ament_cmake</buildtool_depend>

  <depend>std_msgs</depend>
  <depend>rclcpp</depend>
  <depend>gazebo_msgs</depend>
  <depend>std_srvs</depend>
  <depend>unit_8_services_custom_msgs</depend>


  <test_depend>ament_lint_auto</test_depend>
  <test_depend>ament_lint_common</test_depend>

  <export>
    <build_type>ament_cmake</build_type>
  </export>
</package>
```

END Setup: package.xml

Now, let's compile both the **custom server and client**:

Execute in Shell #1

```
$ cd ~/ros2_ws
# We compile only our cpp_unit_7_services package, nothing more
$ colcon build --packages-select cpp_unit_7_services
```

Now, we start the server and call it through the client:

Execute in Shell #1: Launch Service Server

```
$ source .bashrc_ros2
$ ros2 run cpp_unit_7_services custom_service_server_node
```

Execute in Shell #2: Start Client

```
$ source .bashrc_ros2
$ ros2 run cpp_unit_7_services custom_service_client_node
```

Shell #1 Output

```
 [INFO] [custom_service_server]: CustomServiceServer...READY
[INFO] [custom_service_server]: Incoming request
```

radius: 2.300000
[INFO] [custom_service_server]: Incoming request
repetitions: 2

Shell #2 Output

[INFO] [custom_service_client]: Client request->radius: 2.300000
[INFO] [custom_service_client]: Client request->repetitions: 2
[INFO] [custom_service_client]: Result-Success : True

You can also call it through the command line for quick tests:

Execute in Shell #2: Call the Server

```
$ ros2 service call /my_custom_service unit_8_services_custom_msgs/MyCustomServiceMessage '{radius: 5.1, repetitions: 7}'
```

Shell #2 Output

```
waiting for service to become available...
requester: making request: unit_8_services_custom_msgs.srv.MyCustomServiceMessage_Request(radius=5.1, repetitions=7)

response:
unit_8_services_custom_msgs.srv.MyCustomServiceMessage_Response(success=True)
```

- Exercise 8.2 -

Modify the **square_service_server.cpp** and its client that you generated in the previous exercise **8.1** to be able to request different-sized squares and number of repetitions using the **unit_8_services_custom_msgs/MyCustomServiceMessage**.

- End of Exercise 8.2 -

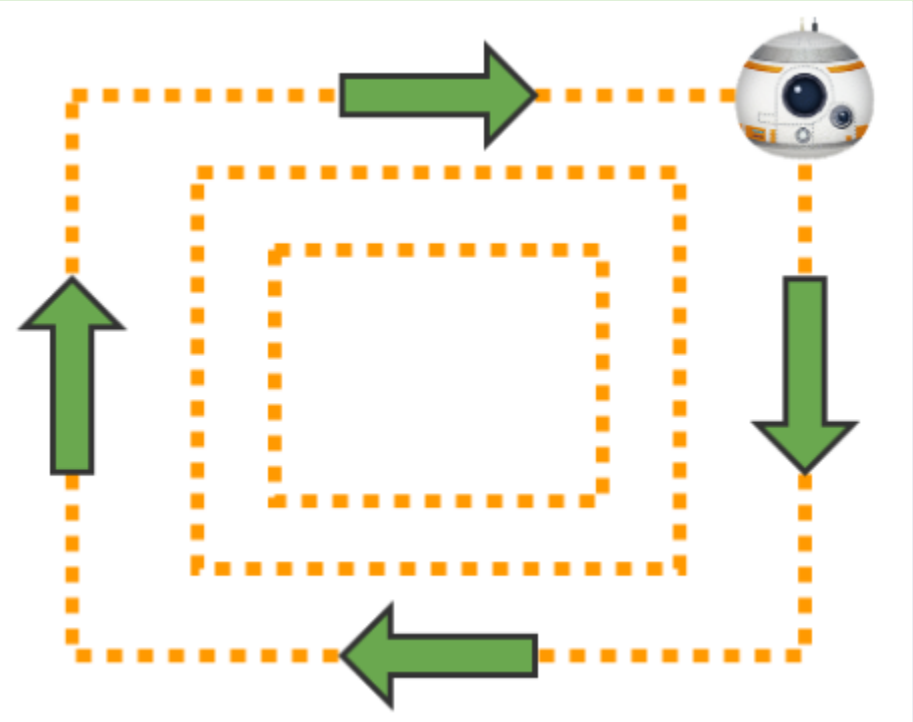


Fig.8.2 - BB8 Dynamic Square Diagram

You should finish PARTS II and III of the rosject before attempting next unit of this course!

ROS2 Basics in 5 days (C++)

Unit 9 ROS2 Actions: Clients

- Summary -

Estimated time to completion: 3 hours

What will you learn with this unit?

- What are actions?
- How to manage the actions of a robot
- How to call an action server

- End of Summary -

1) Did you understand the previous sections about topics and services?
2) Are they clear to you?
3) Did you have a good breakfast today?

If your answers to all of those questions were yes, then you are ready to learn about ROS actions. Otherwise, go back and do not come back until all of those answers are a big YES. You are going to need it...

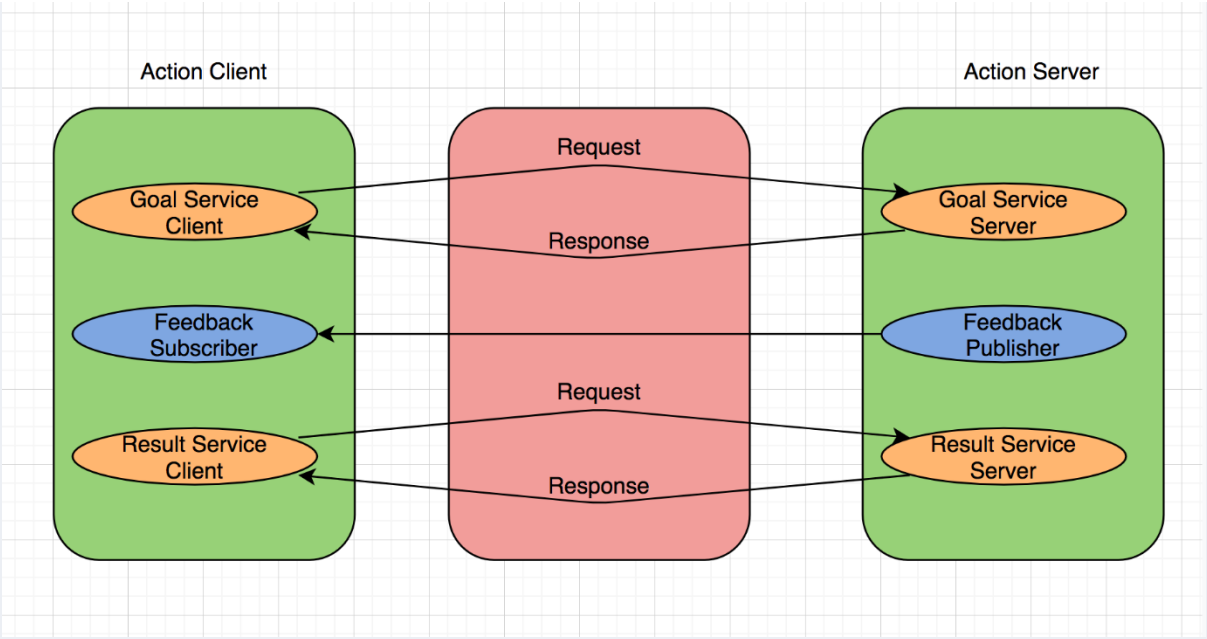
9.1 What are actions?

Actions are very similar to services. When you call an action, you are calling a functionality that another node is providing. Also, actions are based on a client-server model. Just the same as with services.

However, there are 2 main differences between actions and services:

- First, **actions are preemptable**. This means that you can cancel an action while it is being executing.
- Second, **actions provide feedback**. This means that, while the action is being executed, the server can send feedback back to the client.

Below you can see a diagram that describes the workflow of an action.



Action Interface Diagram

Don't worry if you don't fully understand it right now, since it contains many concepts. For now, just keep in mind the below 2 points:

- The node that provides the action functionality has to contain an **action server**. The *action server* allows other nodes to call that action functionality.

- The node that calls to the action functionality has to contain an **action client**. The *action client* allows a node to connect to the *action server* of another node.

Now let's see an action in action (I'm so funny!)

- Notes -

This Unit uses a ROS2 simulation. Therefore, it's not needed to run the ROS1 Bridge.

- End of Notes -

- Exercise 9.1 -

First of all, let's source our ROS2 environment:

Execute in Shell #1

```
$ source .bashrc_ros2
```

Go to a shell and launch the *turtlebot3 action server* with the following command:

Important!! Keep this program running for the rest of the tutorial, since it is the one that provides the action server you are going to use.

Execute in Shell #1, Leave It Running

```
$ ros2 launch turtlebot3_as action_server.launch.py
```

In order to find which actions are available on a robot, you must use the command ***ros2 action list.***

Execute in Shell #2

```
$ ros2 action list
```

Shell #2 Output

```
user ~ $ ros2 action list
...
...
/turtlebot3_as
...
...
```

You can also get data from an specific action with the following command:

Execute in Shell #2

```
ros2 action info /turtlebot3_as
```

Shell #2 Output

```
Action: /turtlebot3_as
Action clients: 0
Action servers: 1
    /t3_action_server
```

Also, if you add the suffix **-t** to the command above, you will get data from the action interface used:

Execute in Shell #2

```
ros2 action info /turtlebot3_as -t
```

Shell #2 Output

```
Action: /turtlebot3_as
Action clients: 0
Action servers: 1
    /t3_action_server [t3_action_msg/action/Move]
```

In our case, as you can see, the **/turtlebot3_as** action uses the interface **t3_action_msg/action/Move**, which means:

```
<pkg_name>/action/<inerface_name>
```

Therefore, our action uses an interface named **Move**, which is defined inside a package named **t3_action_msg**. All action interfaces are defined inside a folder named **action**.
Of course, we can also get more data about this interface with the command:

Execute in Shell #2

```
ros2 interface show t3_action_msg/action/Move
```

Shell #2 Output

```
int32 secs
---
string status
---
string feedback
```

Alright! Now that we have gathered some data about the action server, let's call it!

In order to call an action server, you can use the command ***ros2 action send_goal***. The structure of the command is the following:

```
ros2 action send_goal <action_name> <action_type> <values>
```

Now that we have all the data about the action server, we can complete the command:

Execute in Shell #2

```
ros2 action send_goal /turtlebot3_as t3_action_msg/action/Move "{secs: 5}"
```

Shell #2 Output

```
Waiting for an action server to become available...
```

Sending goal:
    secs: 5

Goal accepted **with** ID: fd252aaa5fee48d595870f0b2a1c9705

Result:
    status: Finished action server. Robot moved during 5 seconds

Goal finished **with** status: SUCCEEDED

Shell #1 Output

[turtlebot3_as-1] [INFO] [1616759072.854181251] [t3_action_server]: Received goal request **with** secs 5
[turtlebot3_as-1] [INFO] [1616759072.855170407] [t3_action_server]: Executing goal
[turtlebot3_as-1] [INFO] [1616759072.855293596] [t3_action_server]: Publish feedback
[turtlebot3_as-1] [INFO] [1616759073.855328070] [t3_action_server]: Publish feedback
[turtlebot3_as-1] [INFO] [1616759074.857161358] [t3_action_server]: Publish feedback
[turtlebot3_as-1] [INFO] [1616759075.855388797] [t3_action_server]: Publish feedback
[turtlebot3_as-1] [INFO] [1616759076.857644650] [t3_action_server]: Publish feedback
[turtlebot3_as-1] [INFO] [1616759077.855392256] [t3_action_server]: Goal succeeded

But wait!! You told me that actions provide feedback, and I can see in the server output that some kind of feedback is being published, so... where's my feedback!!

You are totally right, actions provide feedback. And you can indeed visualize this feedback. However, you need to specify that you want to visualize the feedback when you call the action server.

Try the following command:

Execute in Shell #2

```
ros2 action send_goal -f /turtlebot3_as t3_action_msg/action/Move "{secs: 5}"
```

- Notes -

Note the "**-f**" argument added to the command.

- End of Notes -

Shell #2 Output

Waiting **for** an action server to become available...
Sending goal:
    secs: 5

Goal accepted **with** ID: 9332305ca294430393a3d25156a35947

Feedback:
    feedback: Moving forward...

Feedback:
    feedback: Moving forward...

Feedback:
    feedback: Moving forward...

Feedback:
  feedback: Moving forward...

Feedback:
  feedback: Moving forward...

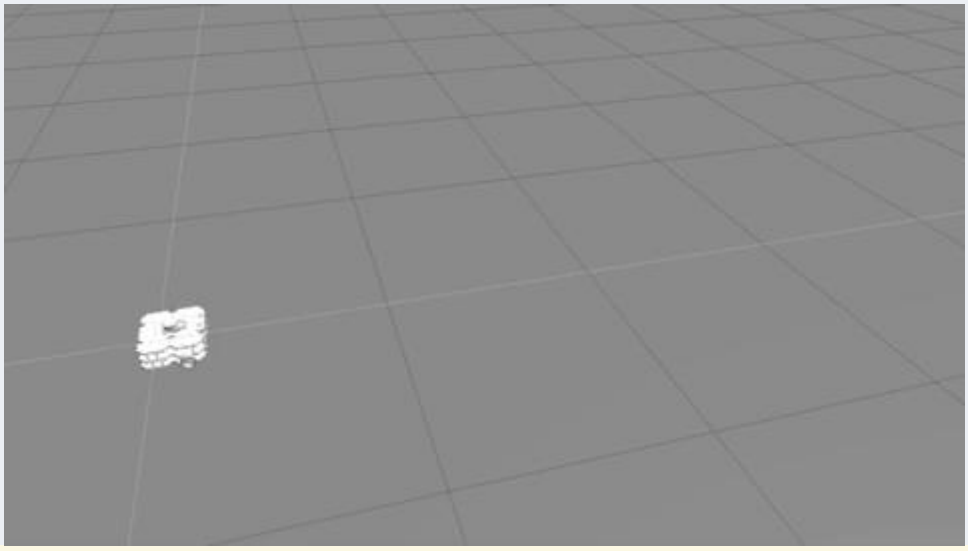
Result:
  status: Finished action server. Robot moved during 5 seconds

Goal finished **with** status: SUCCEEDED

- End of Exercise 9.1 -

- Expected Behavior for Exercise 9.1 -

The robot moves forward for 5 seconds:



- End of Expected Behavior -

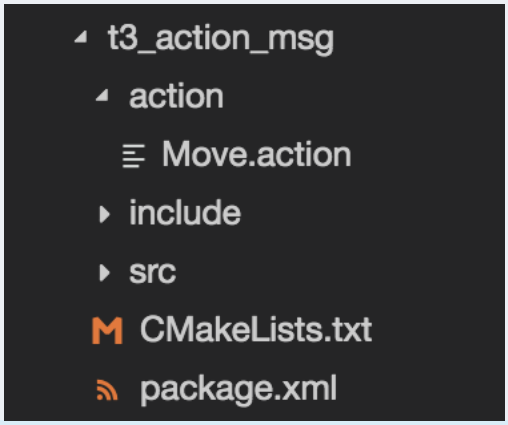
9.2 Calling an action server

The ***turtlebot3_as*** action server is an action that you can call. If you call it, it will start moving the Turtlebot3 robot forward for the amount of seconds specified in the calling message (it is a parameter that you specify in the call).
Calling an action server means sending a message to it. In the same way as with *topics* and *services*, it all works by passing messages around.

- The message of a topic is composed of a single part: the information the topic provides.
- The message of a service has two parts: the request and the response.
- **The message of an action server is divided into three parts: the goal, the result, and the feedback.**

All of the action messages used are defined in the ***action*** directory of their package.

If we check the ***t3_action_msg*** package we will see that it contains a directory called ***action***. Inside that *action* directory, there is a file called ***Move.action***. That is the file that specifies the type of the message that the action uses.



- Notes -

In this case, you can't check this *t3_action_msg* package through the IDE because it's already installed in the system. But don't worry, you will generate your own action interface in the next unit.

- End of Notes -

Let's type again in a Shell the following command to see the message structure:

Execute in Shell #2

```
ros2 interface show t3_action_msg/action/Move
```

Shell #2 Output

```
# goal
int32 secs # the number of seconds the robot will move forward
---
# result
string status # an string indicating the final status when the action ends
---
# feedback
string feedback # an string which indicates the current status of the robot
```

You can see in the previous step how the message is composed of three parts:

goal: Consists of a variable called ***secs*** of type ***int32***. This int32 type is a standard ROS2 message, therefore, it can be found in the std_msgs package. Because it's a standard package of ROS2, it's not needed to indicate the package where the *int32* can be found.

result: Consists of a variable called ***status***, which is of type ***string***, also found in the std_msgs package.

feedback: Consists of a variable called ***feedback*** of type ***string***, also found in the std_msgs package.

You will learn in the second part of this chapter about how to create your own action interfaces. For now, you must only understand that every time you call an action, the message implied contains three parts, and that each part can contain **more than one** variable.

9.2.1 Actions provide feedback

Due to the fact that calling an action server does not interrupt your thread, action servers provide a message called ***the feedback***. The feedback is a message that the action server generates every once in a while to indicate how the action is going (informing the caller of the status of the requested action). It is generated while the action is in progress.

9.2.2 How to call an action server

The way you call an action server is by implementing an ***action client***.

In Exercise 9.1, you created an **action_client** using the command-line tools, with the command ***ros2 action send_goal***. This is very useful for testing or debugging actions. However, calling an action server using the command-line tools has some limitations. For instance, it doesn't allow you to customize the client.

Instead, you will usually have to implement an action client by creating a program.

The following is a self-explanatory example of how to implement an action client that calls the **turtlebot3_as** action server and makes it move forward for 5 seconds.

- Exercise 9.2 -

First of all, let's create a new package where we'll place our action client code.

Execute in Shell #2

```
$ cd ~/ros2_ws/src
$ ros2 pkg create my_action_client --build-type ament_cmake --dependencies rclcpp rclcpp_action t3_action_msg
```

Now, inside the **src** folder of the package, create a new C++ file named **action_client.cpp**. You can paste the below code into the script.

C++ Program: action_client.cpp

```
#include <inttypes.h>
#include <memory>
#include <string>
#include <iostream>

#include "t3_action_msg/action/move.hpp"
#include "rclcpp/rclcpp.hpp"
#include "rclcpp_action/rclcpp_action.hpp"

class T3ActionClient : public rclcpp::Node
{
public:
  using Move = t3_action_msg::action::Move;
  using GoalHandleMove = rclcpp_action::ClientGoalHandle<Move>;

  explicit T3ActionClient(const rclcpp::NodeOptions & node_options = rclcpp::NodeOptions())
  : Node("t3_action_client", node_options), goal_done_(false)
  {
    this->client_ptr_ = rclcpp_action::create_client<Move>(
      this->get_node_base_interface(),
      this->get_node_graph_interface(),
      this->get_node_logging_interface(),
      this->get_node_waitables_interface(),
      "turtlebot3_as");

    this->timer_ = this->create_wall_timer(
      std::chrono::milliseconds(500),
      std::bind(&T3ActionClient::send_goal, this));
```

```
  }

  bool is_goal_done() const
  {
    return this->goal_done_;
  }

  void send_goal()
  {
    using namespace std::placeholders;

    this->timer_->cancel();

    this->goal_done_ = false;

    if (!this->client_ptr_) {
      RCLCPP_ERROR(this->get_logger(), "Action client not initialized");
    }

    if (!this->client_ptr_->wait_for_action_server(std::chrono::seconds(10))) {
      RCLCPP_ERROR(this->get_logger(), "Action server not available after waiting");
      this->goal_done_ = true;
      return;
    }

    auto goal_msg = Move::Goal();
    goal_msg.secs = 5;

    RCLCPP_INFO(this->get_logger(), "Sending goal");

    auto send_goal_options = rclcpp_action::Client<Move>::SendGoalOptions();
    send_goal_options.goal_response_callback =
      std::bind(&T3ActionClient::goal_response_callback, this, _1);
    send_goal_options.feedback_callback =
      std::bind(&T3ActionClient::feedback_callback, this, _1, _2);
    send_goal_options.result_callback =
      std::bind(&T3ActionClient::result_callback, this, _1);
    auto goal_handle_future = this->client_ptr_->async_send_goal(goal_msg, send_goal_options);
  }

private:
  rclcpp_action::Client<Move>::SharedPtr client_ptr_;
  rclcpp::TimerBase::SharedPtr timer_;
  bool goal_done_;

  void goal_response_callback(std::shared_future<GoalHandleMove::SharedPtr> future)
  {
```

```
    auto goal_handle = future.get();
    if (!goal_handle) {
      RCLCPP_ERROR(this->get_logger(), "Goal was rejected by server");
    } else {
      RCLCPP_INFO(this->get_logger(), "Goal accepted by server, waiting for result");
    }
  }

  void feedback_callback(
    GoalHandleMove::SharedPtr,
    const std::shared_ptr<const Move::Feedback> feedback)
  {
    RCLCPP_INFO(
      this->get_logger(),
      "Feedback received: " +
      feedback->feedback);
  }

  void result_callback(const GoalHandleMove::WrappedResult & result)
  {
    this->goal_done_ = true;
    switch (result.code) {
      case rclcpp_action::ResultCode::SUCCEEDED:
        break;
      case rclcpp_action::ResultCode::ABORTED:
        RCLCPP_ERROR(this->get_logger(), "Goal was aborted");
        return;
      case rclcpp_action::ResultCode::CANCELED:
        RCLCPP_ERROR(this->get_logger(), "Goal was canceled");
        return;
      default:
        RCLCPP_ERROR(this->get_logger(), "Unknown result code");
        return;
    }

    RCLCPP_INFO(this->get_logger(), "Result received: " + result.result->status);
    //for (auto res : result.result->status) {
    //  RCLCPP_INFO(this->get_logger(), res);
    //}
  }
};  // class T3ActionClient

int main(int argc, char ** argv)
{
  rclcpp::init(argc, argv);
  auto action_client = std::make_shared<T3ActionClient>();
```

```
  while (!action_client->is_goal_done()) {
    rclcpp::spin_some(action_client);
  }

  rclcpp::shutdown();
  return 0;
}
```

END C++ Program: action_client.cpp

Let's also create a launch file to start our action client. Inside the **my_action_client** package, create a new folder named **launch**. Inside this launch file, create a new file named **action_client.launch.py**, and paste the below code into it:

Launch File: action_client.launch.py

```
from launch import LaunchDescription
import launch_ros.actions


def generate_launch_description():
    return LaunchDescription([
        launch_ros.actions.Node(
            package='my_action_client', executable='action_client', output='screen'),
    ])
```

END Launch File: action_client.launch.py

Great! Next step will be to update the **CMakeLists.txt** file in order to generate the executable node of our action client and install the launch file. Add the below code to the file:

Add to CMakeLists.txt

```
add_executable(action_client src/action_client.cpp)
ament_target_dependencies(action_client
  "rclcpp"
  "rclcpp_action"
  "t3_action_msg")

install(TARGETS
  action_client
  DESTINATION lib/${PROJECT_NAME})

install(DIRECTORY
  launch
  DESTINATION share/${PROJECT_NAME}/
)
```

And finally, just compile the package:

Execute in Shell #1

```
$ cd ~/ros2_ws/src
$ colcon build
$ source instal/setup.bash
```

Awesome! Now you are ready to test your action client. First, of course, you need to start your action server node:

Execute in Shell #1

```
$ ros2 launch turtlebot3_as action_server.launch.py
```

Once the action server is up and running, you can then launch the action client:

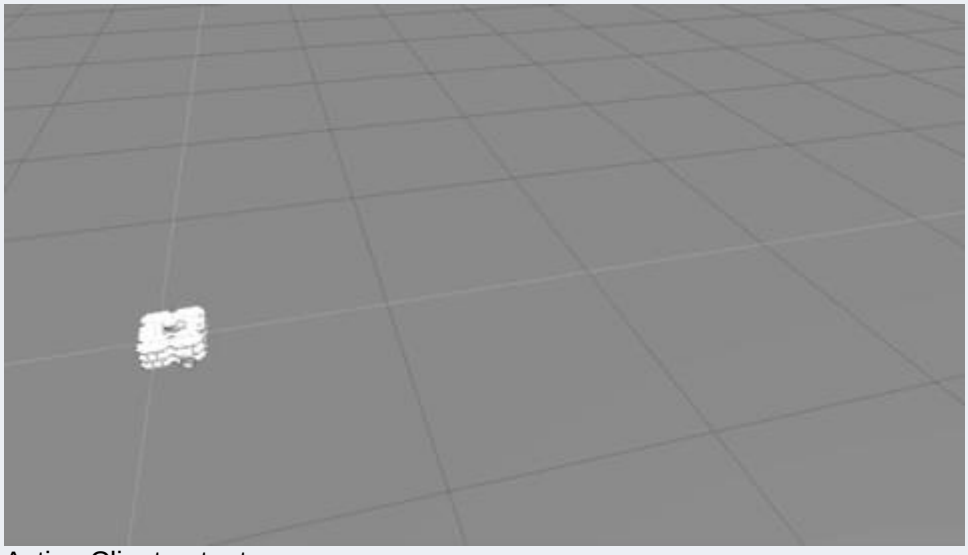
Execute in Shell #2

$ ros2 launch my_action_client action_client.launch.py

- End of Exercise 9.2 -

- Expected Behavior for Exercise 9.2 -

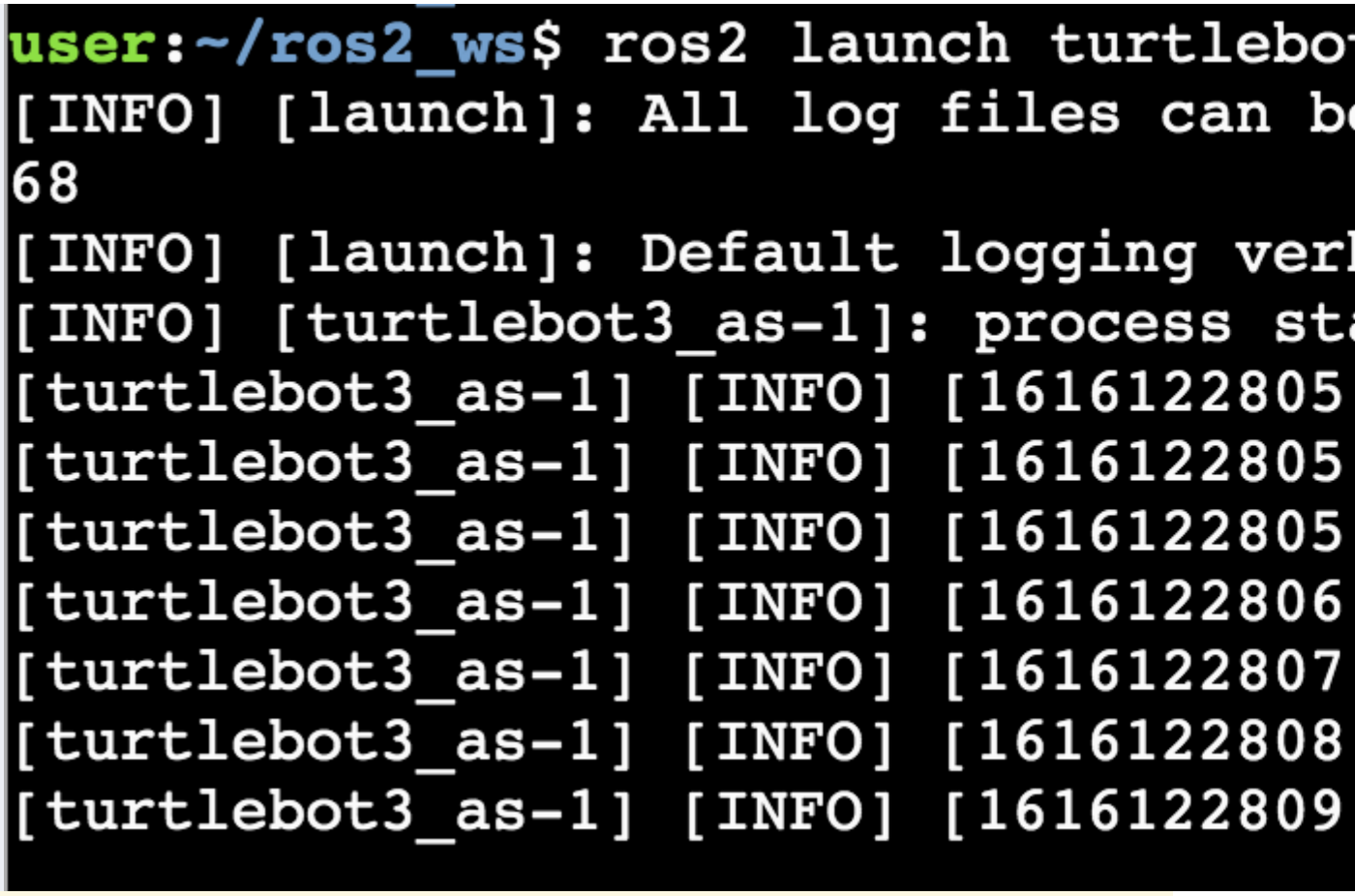
The robot moves forward for 5 seconds:



Action Client output:

```
user:~$ ros2 run my_action_client actio
[INFO] [1616122805.835434657] [t3_actio
[INFO] [1616122805.836188293] [t3_actio
[INFO] [1616122805.836388793] [t3_actio
[INFO] [1616122806.836469115] [t3_actio
[INFO] [1616122807.836488507] [t3_actio
[INFO] [1616122808.836487503] [t3_actio
[INFO] [1616122809.836592030] [t3_actio
5 seconds
```

Action Server output:



- End of Expected Behavior -

9.2.3 Action Client code explanation

Ok, so now you have already executed your first action client, but... what did actually happen? What does the code mean? Let's try to analyze it more in detail.
As always, we start by creating a class, which inherits from the *Node* class:

```
class T3ActionClient : public rclcpp::Node
```

Next we define the 2 most important interfaces that we will use in the action client:

```
using Move = t3_action_msg::action::Move;
using GoalHandleMove = rclcpp_action::ClientGoalHandle<Move>;
```

- **Move**, as you already know, will contain the action interface.
- **GoalHandleMove**, as the name indicates, will be in charge of handling the goal message from *Move*.

Next we are initializing the node of our program. As you can see, the node name is **t3_action_client**:

```
explicit T3ActionClient(const rclcpp::NodeOptions & node_options = rclcpp::NodeOptions())
  : Node("t3_action_client", node_options), goal_done_(false)
```

Right below we create the action client:

```
this->client_ptr_ = rclcpp_action::create_client<Move>(
    this->get_node_base_interface(),
    this->get_node_graph_interface(),
    this->get_node_logging_interface(),
    this->get_node_waitables_interface(),
```

```
    "turtlebot3_as");
```

As you can see, we specify the action interface to use (*Move*), and the action server name (*turtlebot3_as*).
We also create a timer:

```
this->timer_ = this->create_wall_timer(
    std::chrono::milliseconds(500),
    std::bind(&T3ActionClient::send_goal, this));
```

When the time expires (500ms), the timer will call the function **send_goal()**.

Within this **send_goal()** function we are doing several things. Let's start by the beginning:

```
this->goal_done_ = false;

if (!this->client_ptr_) {
  RCLCPP_ERROR(this->get_logger(), "Action client not initialized");
}

if (!this->client_ptr_->wait_for_action_server(std::chrono::seconds(10))) {
  RCLCPP_ERROR(this->get_logger(), "Action server not available after waiting");
  this->goal_done_ = true;
  return;
}
```

First of all we set the **goal_done_** variable to *false*. This variable will help us to know when the action has finished. Also, we check if the action server is up and running, to make sure that it is able to receive goals.
Right below, we specify the goal message to be sent to the server:

```
auto goal_msg = Move::Goal();
goal_msg.secs = 5;

RCLCPP_INFO(this->get_logger(), "Sending goal");
```

As you can see, we are specifying **5 seconds** in the goal message. That's why in the previous exercise the robot moves forward for 5 seconds.

Next we are defining 3 important functions: *goal_response_callback*, *feedback_callback* and *result_callback*. We will see in a moment what they are for.

```
auto send_goal_options = rclcpp_action::Client<Move>::SendGoalOptions();
send_goal_options.goal_response_callback =
  std::bind(&T3ActionClient::goal_response_callback, this, _1);
send_goal_options.feedback_callback =
  std::bind(&T3ActionClient::feedback_callback, this, _1, _2);
send_goal_options.result_callback =
  std::bind(&T3ActionClient::result_callback, this, _1);
```

And finally, we send the goal to the server:

```
auto goal_handle_future = this->client_ptr_->async_send_goal(goal_msg, send_goal_options);
```

When the action server receives and accepts the goal sent from the client, it will send a response back to the client The **goal_response_callback** function is in charge of dealing with this response:

```
void goal_response_callback(std::shared_future<GoalHandleMove::SharedPtr> future)
  {
    auto goal_handle = future.get();
    if (!goal_handle) {
```

```
    RCLCPP_ERROR(this->get_logger(), "Goal was rejected by server");
  } else {
    RCLCPP_INFO(this->get_logger(), "Goal accepted by server, waiting for result");
  }
}
```

As you already know, action servers can send feedback to the client while the action is being executed. This feedback is handled by the **feedback_callback** function:

```
void feedback_callback(
  GoalHandleMove::SharedPtr,
  const std::shared_ptr<const Move::Feedback> feedback)
{
  RCLCPP_INFO(
    this->get_logger(),
    "Feedback received: " +
    feedback->feedback);
}
```

Also, when the action finishes, it sends a result back to the client. This result is handled by the **result_callback** function:

```
void result_callback(const GoalHandleMove::WrappedResult & result)
{
  this->goal_done_ = true;
  switch (result.code) {
    case rclcpp_action::ResultCode::SUCCEEDED:
      break;
    case rclcpp_action::ResultCode::ABORTED:
      RCLCPP_ERROR(this->get_logger(), "Goal was aborted");
      return;
    case rclcpp_action::ResultCode::CANCELED:
      RCLCPP_ERROR(this->get_logger(), "Goal was canceled");
      return;
    default:
      RCLCPP_ERROR(this->get_logger(), "Unknown result code");
      return;
  }

  RCLCPP_INFO(this->get_logger(), "Result received: " + result.result->status);
}
```

Note that we also set **goal_done_** to *true*, indicating that the action has finished.
Finally we find the **main** function:

```
int main(int argc, char ** argv)
{
  rclcpp::init(argc, argv);
  auto action_client = std::make_shared<T3ActionClient>();

  while (!action_client->is_goal_done()) {
    rclcpp::spin_some(action_client);
  }
```

```
  rclcpp::shutdown();
  return 0;
}
```

The logic here is very simple. While the action has not finished (*goal_done_* is *false*), the client will keep running. When the action finishes (*goal_done_* is *true*), the client will shutdown.
In order to check the value of the *goal_done_* variable we use the function **is_goal_done**:

```
bool is_goal_done() const
  {
    return this->goal_done_;
  }
```

And that's it! Now everything makes more sense, doesn't it?

9.3 Perform other tasks while the Action is in progress

As you have seen from the previous exercises, the action code uses a variable named **goal_done_** in order to check the status of the goal.

```
explicit T3ActionClient(const rclcpp::NodeOptions & node_options = rclcpp::NodeOptions())
  : Node("t3_action_client", node_options), goal_done_(false)
```

Also, you can see that we are monitoring this value in our main function:

```
while (!action_client->is_goal_done()) {
    rclcpp::spin_some(action_client);
  }
```

This while loop checks if the value returned by **is_goal_done()** is true or false. If it is false, it means that the action is still in progress, so you can keep doing other things.

- Exercise 9.3 -

In the **my_action_client** package you created previously, create a new C++ script named **action_client2.cpp**.

In this new program, update the C++ script you created in the previous exercise, **action_client.cpp**, so that it now publishes the ***Pose*** data of the robot in a new topic named ***/current_pose*** while the action is being executed (the robot is moving).

IMPORTANT!! Remember that you need to have the ***turtlebot3_as*** action server running (probably in WebShell #1), otherwise this won't work because there will be NO Action Server to be connected to.

- End of Exercise 9.3 -

- Notes for Exercise 9.3 -

The Pose of the robot will be published in a ***geometry_msgs::msg::Pose*** interface.
You can get data about the Pose of the robot by reading from the ***/odom*** topic.

Execute in a Shell

```
ros2 topic echo /odom
```

- End of Notes -

- Solution for Exercise 9.3 -

Please Try to do it by yourself unless you get stuck or need some inspiration. You will learn much more if you fight for each exercise.
Follow this link to open the solutions notebook for Unit 9 Actions Part 1: Actions Part1 Solutions

- End Solution for Exercise 9.3 -

9.3.1 Code Explanation

Let's comment on the most important parts of the solution code. The action client class code basically remains the same, so let's check the updated part:

```
geometry_msgs::msg::Pose pose_msg_;

void topic_callback(const nav_msgs::msg::Odometry::SharedPtr msg)
{
 pose_msg_ = msg->pose.pose;
}

int main(int argc, char ** argv)
{
  rclcpp::init(argc, argv);
  auto action_client = std::make_shared<T3ActionClient>();
  auto publisher = action_client->create_publisher<geometry_msgs::msg::Pose>("current_pose", 10);
  auto subscription = action_client->create_subscription<nav_msgs::msg::Odometry>
      ("/odom", 10, topic_callback);

  while (!action_client->is_goal_done()) {
    rclcpp::spin_some(action_client);
    publisher->publish(pose_msg_);
  }

  rclcpp::shutdown();
  return 0;
}
```

First of all we are defining the **topic_callback** function, which will be in charge of the reading the data from the **/odom** topic and store it into a variable named **pose_msg_**

```
void topic_callback(const nav_msgs::msg::Odometry::SharedPtr msg)
{
  pose_msg_ = msg->pose.pose;
}
```

We can also find the definitions of the publisher and subscriber elements:

```
auto publisher = action_client->create_publisher<geometry_msgs::msg::Pose>("current_pose", 10);
auto subscription = action_client->create_subscription<nav_msgs::msg::Odometry>
  ("/odom", 10, topic_callback);
```

- The publisher will publish ***Pose*** messages to the ***/current_pose*** topic.
- The subscriber will subscribe to the ***/odom*** topic and call the **topic_callback** function each time a new message is published.

Finally, while the action goal has not been completed, we will keep publishing the ***Pose*** message into the ***/current_pose*** topic:

```
while (!action_client->is_goal_done()) {
   rclcpp::spin_some(action_client);
   publisher->publish(pose_msg_);
 }
```

9.5 How does all that work?

You need to understand how the communication inside the actions works. It is not that you are going to use it for programming. However, it will happen that your code will have bugs and you will have to debug it. In order to do proper debugging, you need to understand how the communication between *action servers* and *action clients* works.
As you already know, an *action message* has three parts:

- the goal
- the result
- the feedback

Each one corresponds to a different type of communication between the client and the server. As opposite to ROS1, which used topics for actions communication, in ROS2 there's a combination of topics and services.

The **goal** and the **result** are handled through services, while the **feedback** is handled through a topic.

Look again at the actions communication diagram you saw at the beginning of the chapter.

ROS2 Basics in 5 days (C++)

Unit 10 ROS2 Actions: Servers & interfaces

- Summary -

Estimated time to completion: 2.5 hours

What will you learn with this unit?

- How to create an action server
- How to build your own action interface

- End of Summary -

In the previous lesson, you learned how to **CALL** an action server creating an action client. In this lesson, you are going to learn how to **CREATE** your own action server.

10.1 Writing an action server

- Notes -

This Unit uses a ROS2 simulation. Therefore, it's not needed to run the ROS1 Bridge.

- End of Notes -

In the next exercise you will learn how to create the action server you used in the previous Unit.

- Exercise 10.1 -

First of all, let's create a new package where we'll place our action server code.

Execute in Shell #1

```
$ cd ~/ros2_ws/src
$ ros2 pkg create my_action_server --build-type ament_cmake --dependencies rclcpp rclcpp_action t3_action_msg geometry_msgs
```

Now, inside the **src** folder of the package, create a new C++ file named **action_server.cpp**. You can paste the below code into the script.

C++ Program : action_server.cpp

```
#include <functional>
#include <memory>
#include <thread>

#include "rclcpp/rclcpp.hpp"
#include "rclcpp_action/rclcpp_action.hpp"

#include "t3_action_msg/action/move.hpp"
#include "geometry_msgs/msg/twist.hpp"


class T3ActionServer : public rclcpp::Node
{
public:
  using Move = t3_action_msg::action::Move;
  using GoalHandleMove = rclcpp_action::ServerGoalHandle<Move>;

  explicit T3ActionServer(const rclcpp::NodeOptions & options = rclcpp::NodeOptions())
  : Node("t3_action_server", options)
  {
    using namespace std::placeholders;

    this->action_server_ = rclcpp_action::create_server<Move>(
      this,
      "turtlebot3_as",
      std::bind(&T3ActionServer::handle_goal, this, _1, _2),
      std::bind(&T3ActionServer::handle_cancel, this, _1),
      std::bind(&T3ActionServer::handle_accepted, this, _1));

    publisher_ = this->create_publisher<geometry_msgs::msg::Twist>("cmd_vel", 10);

  }
```

```
private:
  rclcpp_action::Server<Move>::SharedPtr action_server_;
  rclcpp::Publisher<geometry_msgs::msg::Twist>::SharedPtr publisher_;

  rclcpp_action::GoalResponse handle_goal(
    const rclcpp_action::GoalUUID & uuid,
    std::shared_ptr<const Move::Goal> goal)
  {
    RCLCPP_INFO(this->get_logger(), "Received goal request with secs %d", goal->secs);
    (void)uuid;
    return rclcpp_action::GoalResponse::ACCEPT_AND_EXECUTE;
  }

  rclcpp_action::CancelResponse handle_cancel(
    const std::shared_ptr<GoalHandleMove> goal_handle)
  {
    RCLCPP_INFO(this->get_logger(), "Received request to cancel goal");
    (void)goal_handle;
    return rclcpp_action::CancelResponse::ACCEPT;
  }

  void handle_accepted(const std::shared_ptr<GoalHandleMove> goal_handle)
  {
    using namespace std::placeholders;
    // this needs to return quickly to avoid blocking the executor, so spin up a new thread
    std::thread{std::bind(&T3ActionServer::execute, this, _1), goal_handle}.detach();
  }

  void execute(const std::shared_ptr<GoalHandleMove> goal_handle)
  {
    RCLCPP_INFO(this->get_logger(), "Executing goal");
    const auto goal = goal_handle->get_goal();
    auto feedback = std::make_shared<Move::Feedback>();
    auto & message = feedback->feedback;
    message = "Starting movement...";
    auto result = std::make_shared<Move::Result>();
    auto move = geometry_msgs::msg::Twist();
    rclcpp::Rate loop_rate(1);

    for (int i = 0; (i < goal->secs) && rclcpp::ok(); ++i) {
      // Check if there is a cancel request
      if (goal_handle->is_canceling()) {
        result->status = message;
        goal_handle->canceled(result);
        RCLCPP_INFO(this->get_logger(), "Goal canceled");
        return;
```

```
      }
      // Move robot forward and send feedback
      message = "Moving forward...";
      move.linear.x = 0.3;
      publisher_->publish(move);
      goal_handle->publish_feedback(feedback);
      RCLCPP_INFO(this->get_logger(), "Publish feedback");

      loop_rate.sleep();
    }

    // Check if goal is done
    if (rclcpp::ok()) {
      result->status = "Finished action server. Robot moved during 5 seconds";
      move.linear.x = 0.0;
      publisher_->publish(move);
      goal_handle->succeed(result);
      RCLCPP_INFO(this->get_logger(), "Goal succeeded");
    }
  }
};  // class T3ActionServer

int main(int argc, char ** argv)
{
  rclcpp::init(argc, argv);

  auto action_server = std::make_shared<T3ActionServer>();

  rclcpp::spin(action_server);

  rclcpp::shutdown();
  return 0;
}
```

END C++ Program : action_server.cpp

Let's also create a launch file to start our action server. Inside the **my_action_server** package, create a new folder named **launch**. Inside this launch file, create a new file named **action_server.launch.py**, and paste the below code into it:

Launch File : action_server.launch.py

```
from launch import LaunchDescription
import launch_ros.actions


def generate_launch_description():
    return LaunchDescription([
        launch_ros.actions.Node(
            package='my_action_server', executable='action_server', output='screen'),
    ])
```

END Launch File : action_server.launch.py

Great! Next step will be to update the **CMakeLists.txt** file in order to generate the executable node of our action server and install the launch file. Add the below code to the file:

Add to CMakeLists.txt

```
add_executable(action_server src/action_server.cpp)
ament_target_dependencies(action_server
  "rclcpp"
  "rclcpp_action"
  "t3_action_msg"
  "geometry_msgs")

install(TARGETS
  action_server
  DESTINATION lib/${PROJECT_NAME})

install(DIRECTORY
  launch
  DESTINATION share/${PROJECT_NAME}/
)
```

And finally, just compile the package:

Execute in Shell #1

```
$ cd ~/ros2_ws/src
$ colcon build
$ source instal/setup.bash
```

Let's now launch our action server:

Execute in Shell #1

```
$ ros2 launch my_action_server action_server.launch.py
```

And launch the action client next:

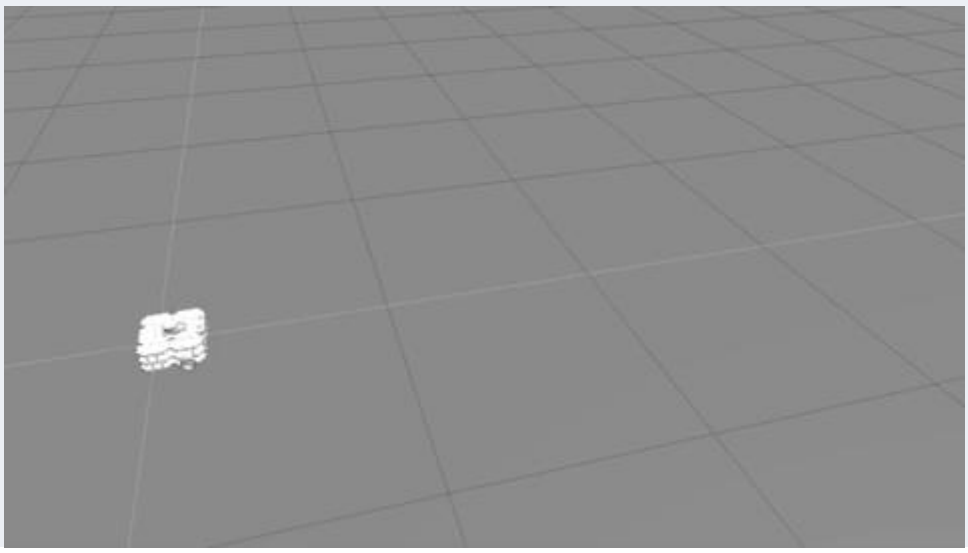
Execute in Shell #2

```
$ ros2 launch my_action_client action_client.launch.py
```

- End of Exercise 10.1 -

- Expected Behavior for Exercise 10.1 -

The robot moves forward for 5 seconds:



- End of Expected Behavior -

10.1.1 Action Server code explanation

Ok, so now that you have already created your first action server, let's try to analyze the code more in detail.
As always, we start by creating a class, which inherits from the *Node* class:

```
class T3ActionServer : public rclcpp::Node
```

Next we define the 2 most important interfaces that we will use in the action server:

```
using Move = t3_action_msg::action::Move;
using GoalHandleMove = rclcpp_action::ServerGoalHandle<Move>;
```

- **Move**, as you already know, will contain the action interface.
- **GoalHandleMove**, as the name indicates, will be in charge of handling the goal message from *Move*.

Next we are initializing the node of our program. As you can see, the node name is **t3_action_server**:

```
explicit T3ActionServer(const rclcpp::NodeOptions & options = rclcpp::NodeOptions())
  : Node("t3_action_server", options)
```

Right below we create the action server:

```
this->action_server_ = rclcpp_action::create_server<Move>(
    this,
    "turtlebot3_as",
    std::bind(&T3ActionServer::handle_goal, this, _1, _2),
    std::bind(&T3ActionServer::handle_cancel, this, _1),
    std::bind(&T3ActionServer::handle_accepted, this, _1));
```

As you can see, in order to create an action server we need to define 5 things:

- The action interface. In this case, **Move**.
- The action name. In this case, **turtlebot3_as**.
- A callback function for handling the goals sent from the client: **handle_goal**.
- A callback function for handling the preemption (cancellation) of goals: **handle_cancel**.
- A callback function for handling the accepted goals: **handle_accepted**.

Right below we are also defining a publisher, for publishing *Twist* messages into the */cmd_vel* topic.

```
publisher_ = this->create_publisher<geometry_msgs::msg::Twist>("cmd_vel", 10);
```

Next we find the **handle_goal** function:

```
rclcpp_action::GoalResponse handle_goal(
   const rclcpp_action::GoalUUID & uuid,
   std::shared_ptr<const Move::Goal> goal)
 {
   RCLCPP_INFO(this->get_logger(), "Received goal request with secs %d", goal->secs);
   (void)uuid;
   return rclcpp_action::GoalResponse::ACCEPT_AND_EXECUTE;
 }
```

In our case, we are just accepting ALL the goals for simplification. However, bear in mind that you can filter here the incoming goals (for instance, don't accept goals which contain negative seconds).
Next we find the **handle_cancel** function:

```
rclcpp_action::CancelResponse handle_cancel(
  const std::shared_ptr<GoalHandleMove> goal_handle)
{
  RCLCPP_INFO(this->get_logger(), "Received request to cancel goal");
  (void)goal_handle;
  return rclcpp_action::CancelResponse::ACCEPT;
}
```

Again, here we are just accepting all cancellation requests. However, we could also reject a cancellation request from the client (for instance, reject the cancellation request if the robot has already started moving).
Next we find the **handle_accepted** function:

```
void handle_accepted(const std::shared_ptr<GoalHandleMove> goal_handle)
{
  using namespace std::placeholders;
  // this needs to return quickly to avoid blocking the executor, so spin up a new thread
  std::thread{std::bind(&T3ActionServer::execute, this, _1), goal_handle}.detach();
}
```

As you can see, here we are generating a new thread (**execute**), which will perform the actual functionality of the action.

And finally we have the **execute** function, which is the one that will do the real work. Let's review it step by step. First of all we define all the variables the we will be using:

```
const auto goal = goal_handle->get_goal();
auto feedback = std::make_shared<Move::Feedback>();
auto & message = feedback->feedback;
message = "Starting movement...";
auto result = std::make_shared<Move::Result>();
auto move = geometry_msgs::msg::Twist();
rclcpp::Rate loop_rate(1);
```

We have:

- The **goal** variable, which will contain the goal message sent by the client.
- The **feedback** variable, which will contain the feedback message that we will send back to the client.
- The **result** variable, which will contain the result message that we will send to the client when the action finishes.
- The **move** variable, which will contain the Twist message used to send velocities to the robot.
- A **loop_rate** variable of 1Hz (1 second).

Next, we start a *for* loop that will keep executing until the variable **i** reaches the number of seconds specified in the goal message.

```
for (int i = 0; (i < goal->secs) && rclcpp::ok(); ++i) {
```

For instance, if we set 5 seconds in the goal message, this loop will be executed once per second, during 5 seconds.
Next we check if the action has been cancelled. If it's cancelled, we will terminate the action.

```
if (goal_handle->is_canceling()) {
      result->status = message;
      goal_handle->canceled(result);
      RCLCPP_INFO(this->get_logger(), "Goal canceled");
      return;
    }
```

If the action is not cancelled, we will send a *Twist* message to the robot in order to move it forward at an speed of 0.3 m/s, and we will send a feedback message back to the client with the string *"Moving forward..."*.

```
message = "Moving forward...";
move.linear.x = 0.3;
publisher_->publish(move);
goal_handle->publish_feedback(feedback);
RCLCPP_INFO(this->get_logger(), "Publish feedback");
loop_rate.sleep();
```

At the end of the *execute* function, we check if the goal has been completed:

```
if (rclcpp::ok()) {
    result->status = "Finished action server. Robot moved during 5 seconds";
    move.linear.x = 0.0;
    publisher_->publish(move);
    goal_handle->succeed(result);
    RCLCPP_INFO(this->get_logger(), "Goal succeeded");
```

If the goal has been accomplished, we will do 2 things:

- We stop the robot by sending a velocity of 0.
- We fill the *result* message and send it back to the client.

Finally we can find the **main** function:

```
int main(int argc, char ** argv)
{
 rclcpp::init(argc, argv);

 auto action_server = std::make_shared<T3ActionServer>();

 rclcpp::spin(action_server);

 rclcpp::shutdown();
 return 0;
}
```

As you can see, it's very simple. We just spin the action server node so that we keep the action server up and running.
And that's it! Now everything makes more sense, doesn't it?

10.2 Create your own action interface

To create your own action interface you have to complete the following 3 steps:

1. Create an *action* directory within your package.

2. Create your *Name.action* action message file.

- The Name of the action message file will determine later the name of the classes to be used in the **action server** and/or **action client**. ROS2 convention indicates that the name has to be camel-case.

- Remember the Name.action file has to contain three parts, each part separated by three hyphens.

```
#goal
package_where_message_is/message_type goal_var_name
---
#result
package_where_message_is/message_type result_var_name
---
#feedback
package_where_message_is/message_type feedback_var_name
```

- If you do not need one part of the message (for example, you don't need to provide feedback), then you can leave that part empty. But you **must always specify the hyphen separtors**.

3. Modify the *CMakeLists.txt* and *package.xml* files to include action message compilation. Read the detailed description below.

10.2.1 Prepare CMakeLists.txt and package.xml files

You have to edit two files in the package, in the same way that we explained for topics and services:

- CMakeLists.txt
- package.xml

Modification of CMakeLists.txt

In order to create a new action interface, you need to call the **rosidl_generate_interfaces** function in your CMakeLists.txt file. To call this function, you have to add the below snippet to your CMakeLists.txt file:

```
rosidl_generate_interfaces(${PROJECT_NAME}
  "action/Name.action"
)
```

Within the function, you have to specify the name of your action interface file.
Also, for being able to generate action interfaces, you have to make sure that you have access to the following packages:

- **rosidl_default_generators**

- **action_msgs**

For this, you have to also add the below line to your CMakeLists.txt file:

```
find_package(action_msgs REQUIRED)
find_package(rosidl_default_generators REQUIRED)
```

As a reference, the *CMakeLists.txt* file of the **t3_action_msg** package, which contains the **Move** action interface, looks like this:

CMakeLists.txt of t3_action_msg package

```
cmake_minimum_required(VERSION 3.5)
project(t3_action_msg)

# Default to C99
if(NOT CMAKE_C_STANDARD)
  set(CMAKE_C_STANDARD 99)
endif()

# Default to C++14
if(NOT CMAKE_CXX_STANDARD)
  set(CMAKE_CXX_STANDARD 14)
endif()

if(CMAKE_COMPILER_IS_GNUCXX OR CMAKE_CXX_COMPILER_ID MATCHES "Clang")
  add_compile_options(-Wall -Wextra -Wpedantic)
endif()

# find dependencies
find_package(ament_cmake REQUIRED)
find_package(rclcpp REQUIRED)
find_package(action_msgs REQUIRED)
find_package(rosidl_default_generators REQUIRED)

rosidl_generate_interfaces(${PROJECT_NAME}
  "action/Move.action"
)

if(BUILD_TESTING)
  find_package(ament_lint_auto REQUIRED)
  # the following line skips the linter which checks for copyrights
  # uncomment the line when a copyright and license is not present in all source files
  #set(ament_cmake_copyright_FOUND TRUE)
  # the following line skips cpplint (only works in a git repo)
  # uncomment the line when this package is not in a git repo
  #set(ament_cmake_cpplint_FOUND TRUE)
  ament_lint_auto_find_test_dependencies()
endif()

ament_package()
```

Modification of package.xml

In the *package.xml* file you have to make sure to have dependencies for the following packages:

- **action_msgs**
- **rosidl_default_generators**

```
<depend>action_msgs</depend>
<depend>rosidl_default_generators</depend>
```

Also, you have to specify **rosidl_interface_packages** package as member_of_group:

```
<member_of_group>rosidl_interface_packages</member_of_group>
```

As a reference, the *CMakeLists.txt* file of the **t3_action_msg** package, which contains the **Move** action interface, looks like this:

package.xml of t3_action_msg package

```
<?xml version="1.0"?>
<?xml-model href="http://download.ros.org/schema/package_format3.xsd" schematypens="http://www.w3.org/2001/XMLSchema"?>
<package format="3">
  <name>t3_action_msg</name>
  <version>0.0.0</version>
  <description>TODO: Package description</description>
  <maintainer email="user@todo.todo">user</maintainer>
  <license>TODO: License declaration</license>

  <buildtool_depend>ament_cmake</buildtool_depend>

  <depend>rclcpp</depend>
  <depend>action_msgs</depend>
  <depend>rosidl_default_generators</depend>

  <member_of_group>rosidl_interface_packages</member_of_group>

  <test_depend>ament_lint_auto</test_depend>
  <test_depend>ament_lint_common</test_depend>

  <export>
    <build_type>ament_cmake</build_type>
  </export>
</package>
```

Finally, when everything is correctly set up, you just have to compile:

```
$ colcon build
$ source instal/setup.bash
```

To verify that your action interface has been created correctly, you can use the following command:

```
ros2 interface show <pkg_name>/action/<inerface_name>
```

- Notes -

Note that you haven't imported the **std_msgs** package anywhere. But you can use the messages declared there in your action interfaces. That's because this package forms part of the core ROS2 files, so therefore, it's embedded in the compilation protocols, and no declaration of use is needed.

- End of Notes -

10.3 Actions Mini Project

- Mini Project -

Create a new package with an action server which uses custom action interface to move the robot. It will work like this:

- The new action server will receive 3 *string* as a goal: FORWARD, BACKWARD or STOP.
- When the action server receives the FORWARD string, the robot will move forward.
- When the action server receives the BACKWARD string, the robot will move backward.
- When the action server receives the STOP string, the robot will stop any movement.
- As a feedback, it publishes once a second what action is taking place.
- When the action finishes, the result will return nothing.

- End of Mini Project -

- Notes for Mini Project -

- You need to create a new action interface with the following structure:

```
string goal
---
---
string feedback
```

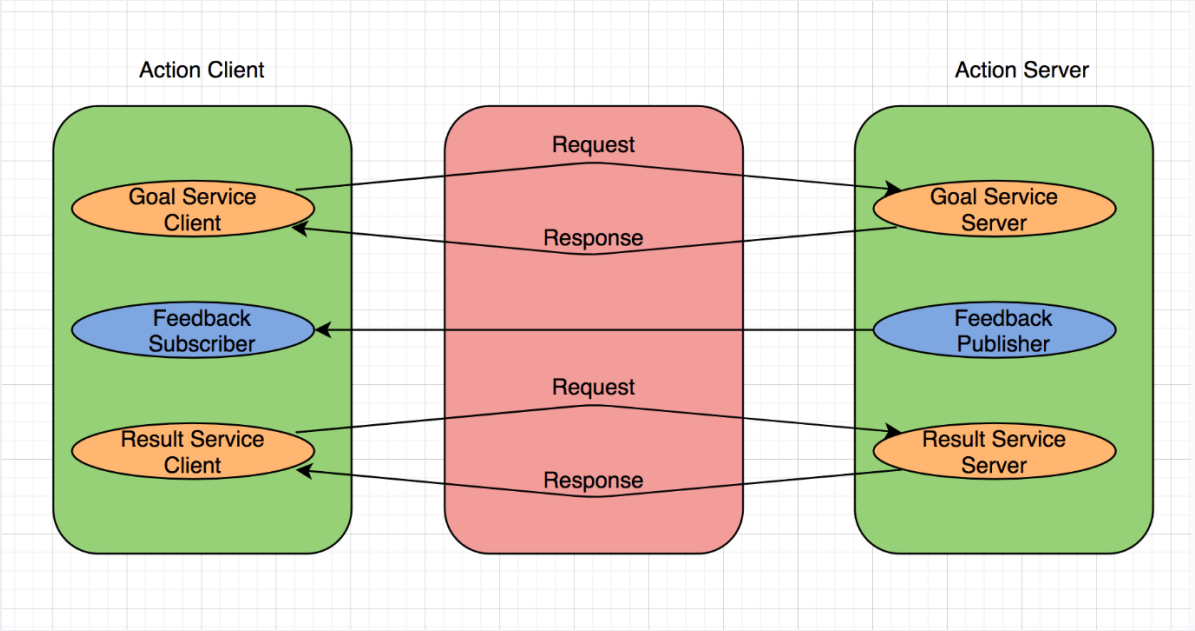
- The name of the package where you'll place all the code related to the project will be **actions_project**.
- The name of the launch file that will start your Action Server will be **action_custom_msg.launch.py**.
- The name of the action will be **/action_custom_msg_as**.
- The name of your Action message file will be **Project.action**.

- End of Notes -

- Solution for Mini Project -

The Construct

Please Try to do it by yourself unless you get stuck or need some inspiration. You will learn much more if you fight for each exercise.
Follow this link to open the solutions notebook for Unit 10 Actions Part 2: Actions Part2 Solutions
- End of Solution -



Action Interface Diagram

So, whenever an action server is called, the sequence of steps is as follows:

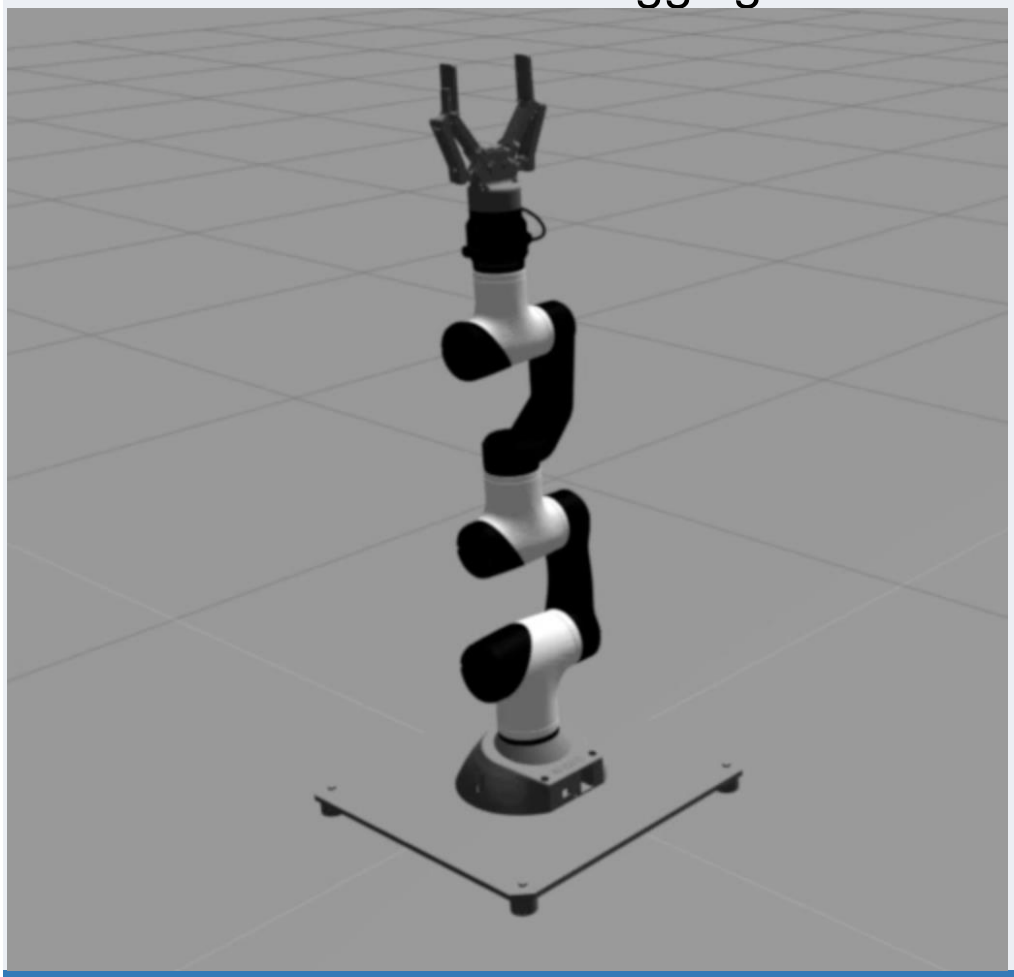
1.- First of all, the **Goal Service Client** from the **Action Client** sends a ***request*** to the **Goal Service Server** of the **Action Server**. Then, the Action Server evaluates the goal and sends a ***response*** to the Action Client indicating if the goal has been accepted or rejected.

2.- If the goal is accepted, then the **Result Service Client** from the **Action Client** sends a ***request*** to the **Result Service Server** of the **Action Server**. At this point, the action will start to be executed (in our example, the robot starts to move).

3.- Meanwhile the action is being executed, the **Feedback Publisher** from the **Action Server** will start to send messages to the **Feedback Subscriber** of the **Action Client**.

4.- Finally, when the goal has been completed, the **Result Service Server** from the **Action Server** sends a ***response*** to the **Result Service Client** of the **Action Client**.

Unit 11 Debugging Tools



- Summary -

Estimated time to completion: 1.5 hours

What will you learn with this unit?

- Add Debugging ROS logs
- Basic use of RViz2 debugging tool

- End of Summary -

One of the most difficult, but important, parts of robotics is: **knowing how to turn your ideas and knowledge into real projects**. There is a constant in robotics projects: **nothing works as in theory**. Reality is much more complex and, therefore, you need tools to discover what is going on and find where the problem lies. That's why debugging and visualization tools are essential in robotics, especially when working with complex data formats, such as **images**, **laser-scans**, **pointclouds,** or **kinematic data**. Examples are shown in {Fig-11.i} and {Fig-11.ii}.

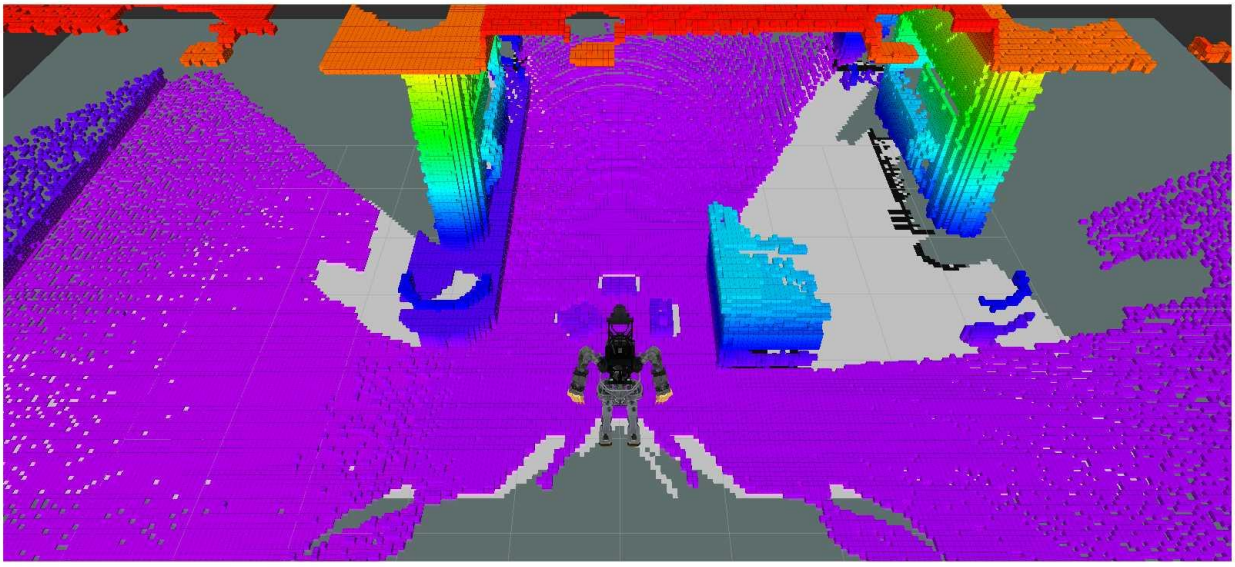


Fig.11.i - Atlas Laser

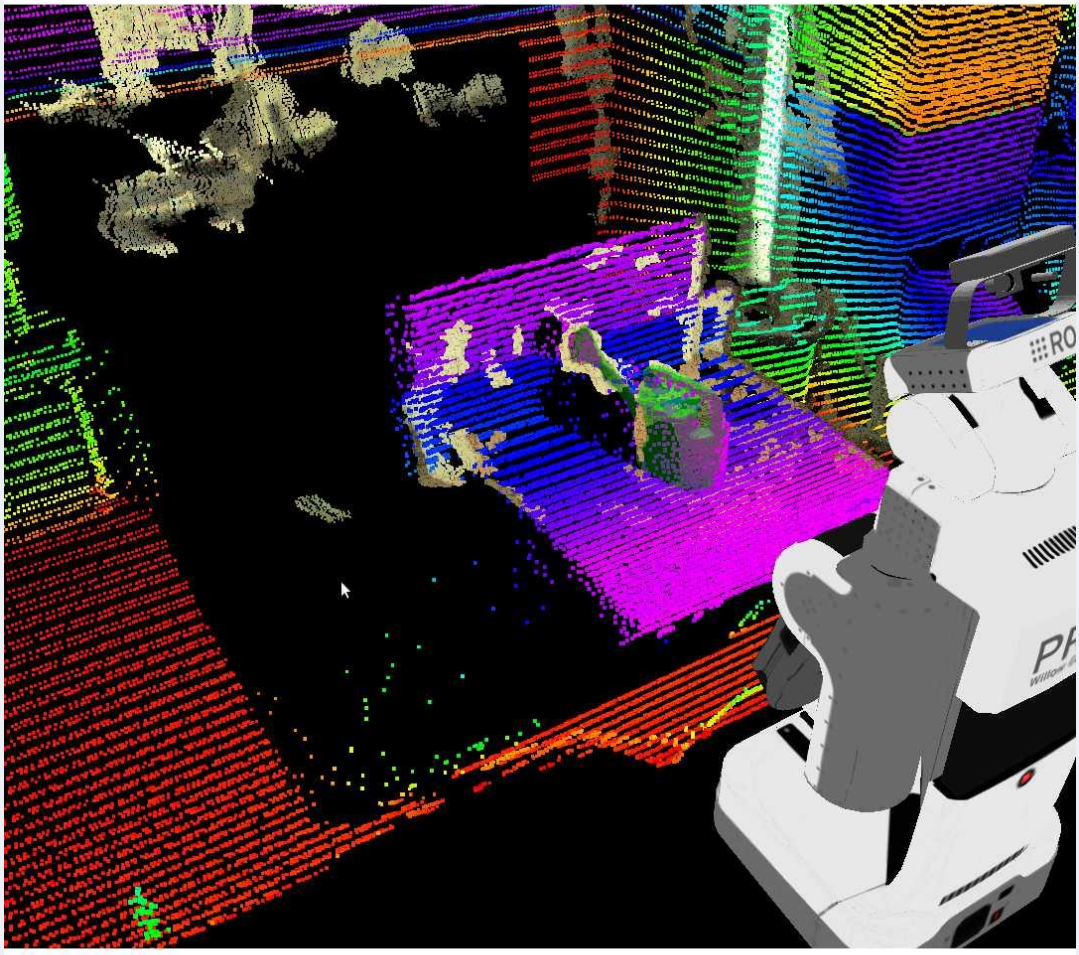


Fig.11.ii - PR2 Laser and PointCloud

So, here you will be presented with the most important tools for debugging your code and visualizing what is really happening in your robot system.

11.1 ROS Debugging Messages

Logs allow you to print them on the screen, but also to store them in the ROS framework, so you can classify, sort, filter, or something else.
In logging systems, there are always levels of logging, as shown in the image below. In ROS2 logs case, there are **five** levels. Each level includes deeper levels. So, for example, if you use **Error** level, all the messages for **Error** and **Fatal** will be shown. If your level is **Warning**, then all the messages for levels **Warning**, **Error**, and **Fatal** will be shown.

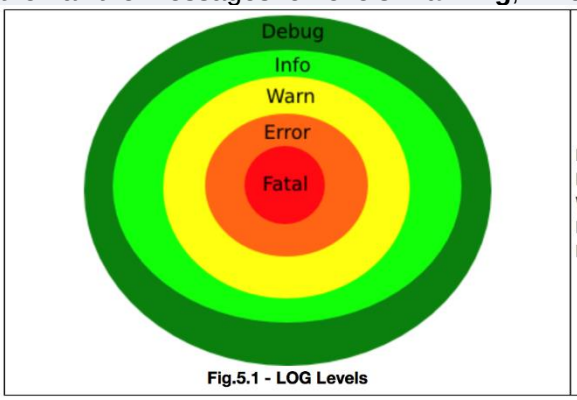


Fig.5.1 - LOG Levels

```
DEBUG ==> RCLCPP_DEBUG("This is a Debug Log");
INFO ==> RCLCPP_INFO("This is an Info Log");
WARNING ==> RCLCPP_WARN("This is a Warning Log");
ERROR ==> RCLCPP_ERROR("This is an Error Log");
FATAL ==> RCLCPP_FATAL("This is a Fatal Log");
```

Run the following C++ code:

- Exercise 11.1 -

- Create a new package named **logs_test**. When creating the package, add **rclcpp** as dependencies. .
- Inside the src folder of the package, create a new file named **logger_example.cpp**. Inside this file, copy the contents of logger_example.cpp
- Create a launch file for launching this code.
- Do the necessary modifications to your **CMakeLists.txt** file, and compile the package.
- Execute the launch file to run your executable.

C++ Program: logger_example.cpp

```
#include "rclcpp/rclcpp.hpp"

int main(int argc, char * argv[])
{
  rclcpp::init(argc, argv);
  auto node = rclcpp::Node::make_shared("log_demo");
  rclcpp::WallRate loop_rate(0.5);
  rcutils_logging_set_logger_level(node->get_logger().get_name(), RCUTILS_LOG_SEVERITY_DEBUG);

  while (rclcpp::ok()) {

    RCLCPP_DEBUG(node->get_logger(), "There is a missing droid");
    RCLCPP_INFO(node->get_logger(), "The Emperor's cappuccino is done");
    RCLCPP_WARN(node->get_logger(), "Help me Obi-Wan Kenobi, you're my only hope");
    RCLCPP_ERROR(node->get_logger(), "The rebels are breaking our defenses");
    RCLCPP_FATAL(node->get_logger(), "The DeathStar Is EXPLODING");
    rclcpp::spin_some(node);
```

```
    loop_rate.sleep();

  }
  rclcpp::shutdown();
  return 0;
}
```

END C++ Program: logger_example.cpp

When compiling, you will probably see the following warning:

```
--- stderr: logs_test
/home/user/ros2_ws/src/logs_test/src/logger_example.cpp: In function 'int main(int, char**)':
/home/user/ros2_ws/src/logs_test/src/logger_example.cpp:9:35: warning: ignoring return value of 'rcutils_ret_t rcutils_logging_set_logger_level(const char*, int)', declared with attribute warn_unused_result [-Wunused-result]
   rcutils_logging_set_logger_level(node->get_logger().get_name(), RCUTILS_LOG_SEVERITY_ERROR);
   ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
---
```

Just ignore it, it won't affect the exercise.
You should see all of the ROS2 logs in the current nodes, running in the system.

```
user:~/ros2_ws$ ros2 run logs_test logs_test_node
[DEBUG] [log_demo]: There is a missing droid
[INFO] [log_demo]: The Emperors Capuchino is done
[WARN] [log_demo]: Help me Obi-Wan Kenobi, you're my only hope
[ERROR] [log_demo]: The rebels are breaking our defenses
[FATAL] [log_demo]: The DeathStar Is EXPLODING
[DEBUG] [log_demo]: There is a missing droid
[INFO] [log_demo]: The Emperors Capuchino is done
[WARN] [log_demo]: Help me Obi-Wan Kenobi, you're my only hope
[ERROR] [log_demo]: The rebels are breaking our defenses
[FATAL] [log_demo]: The DeathStar Is EXPLODING
[DEBUG] [log_demo]: There is a missing droid
[INFO] [log_demo]: The Emperors Capuchino is done
[WARN] [log_demo]: Help me Obi-Wan Kenobi, you're my only hope
[ERROR] [log_demo]: The rebels are breaking our defenses
[FATAL] [log_demo]: The DeathStar Is EXPLODING
```

- End of Exercise 11.1 -

- Exercise 11.2 -

1- Change the LOG level in the previous code {logger_example.cpp} and see how the different messages are printed or not, depending on the level selected.

2- Remember that you will need to recompile the package each time you make a modification in the code.

3- The line where you change the LOG level is the following:

```
rcutils_logging_set_logger_level(node->get_logger().get_name(), RCUTILS_LOG_SEVERITY_<LOG_LEVEL>);
```

- End of Exercise 11.2 -

11.2 Visualize Complex data and RViz2

And here you have it. The **HollyMolly!** The Millenium Falcon! The most important tool for ROS debugging....**RVIZ2**.

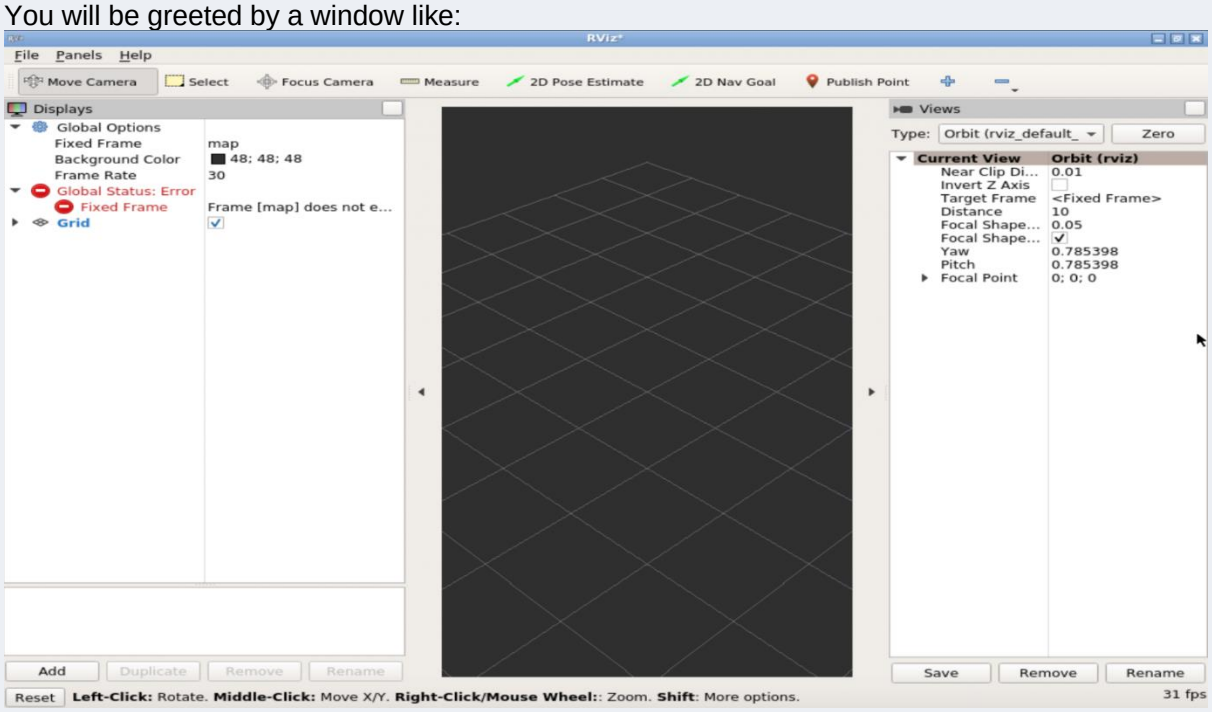
RVIZ is a tool that allows you to visualize *Images, PointClouds, Lasers, Kinematic Transformations, RobotModels*... The list is endless. You can even define your own markers. It's one of the reasons why ROS was so greatly accepted. Before RVIZ, it was really difficult to know what the Robot was perceiving. And that's the main concept:

RVIZ is **NOT** a simulation. I repeat: It's **NOT** a simulation.
RVIZ is a representation of what is being published in the topics, by the simulation or the real robot.
RVIZ is a really complex tool and it would take you a whole course just to master it. Here, you will get a glimpse of what it can give you.
1- Type the following command into WebShell #1:

Execute in Shell #1

```
$ source .bashrc_ros2
$ rviz2
```

2- Then, go to the graphical interface to see the RViz2 GUI:
You will be greeted by a window like:



RViz2 Starting Window

- Notes -

Note: In case you don't see the lower part of RViz2 (the Add button, etc.), double-click at the top of the window to maximize it. Then, you'll see it properly.

- End of Notes -

You need only to be concerned about a few elements to start enjoying RVIZ.

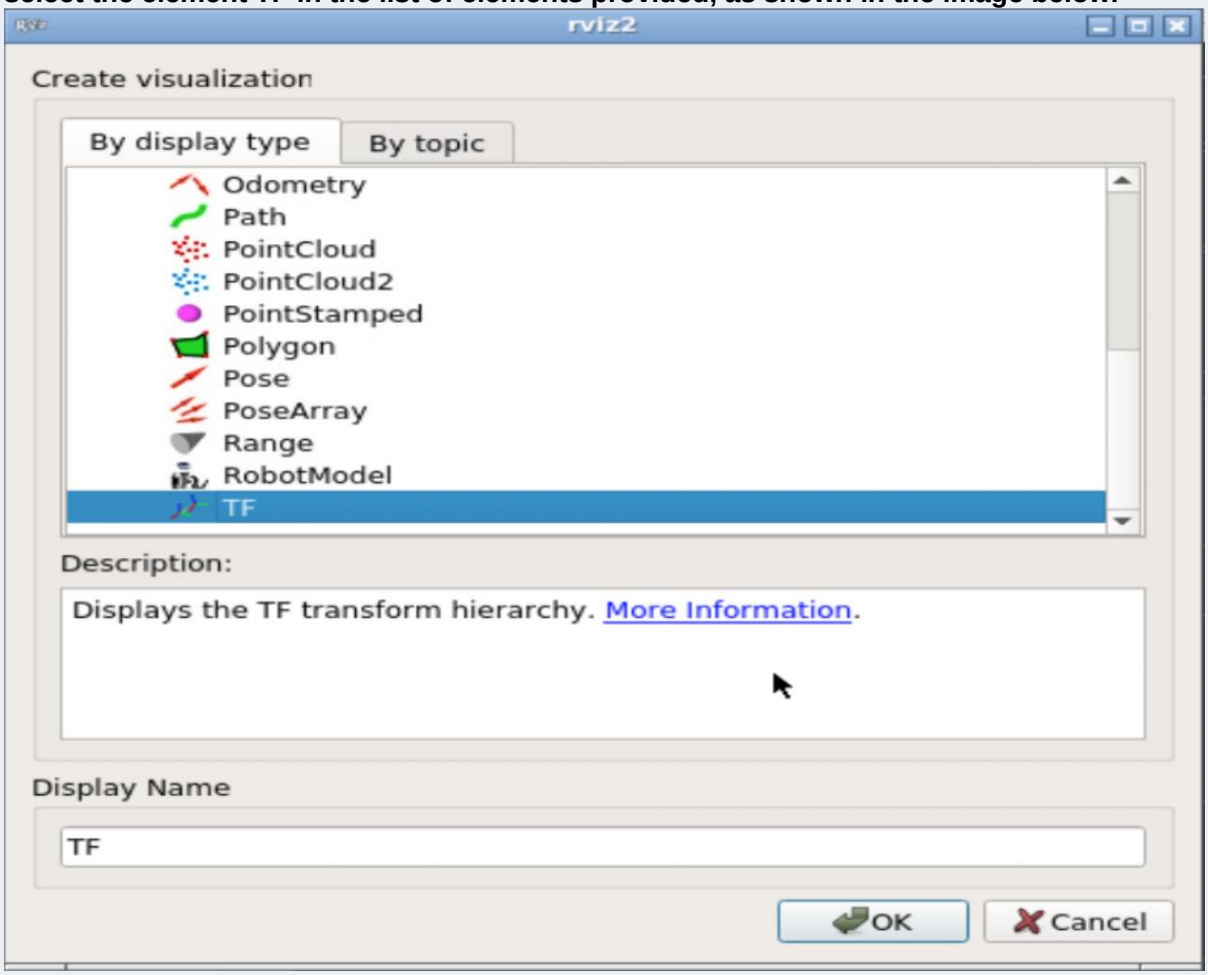
- **Central Panel**: Here is where all the magic happens. This is where the data will be shown. It's a 3D space that you can rotate (LEFT-CLICK PRESSED), translate (CENTER MOUSE BUTTON PRESSED), and zoom in/out (LEFT-CLICK PRESSED).

- **LEFT Displays Panel**: Here is where you manage/configure all the elements that you wish to visualize in the central panel. You only need to use two elements:

- In *Global Options*, you have to select the **Fixed Frame** that suits you for the visualization of the data. It is the reference frame from which all the data will be referred to.

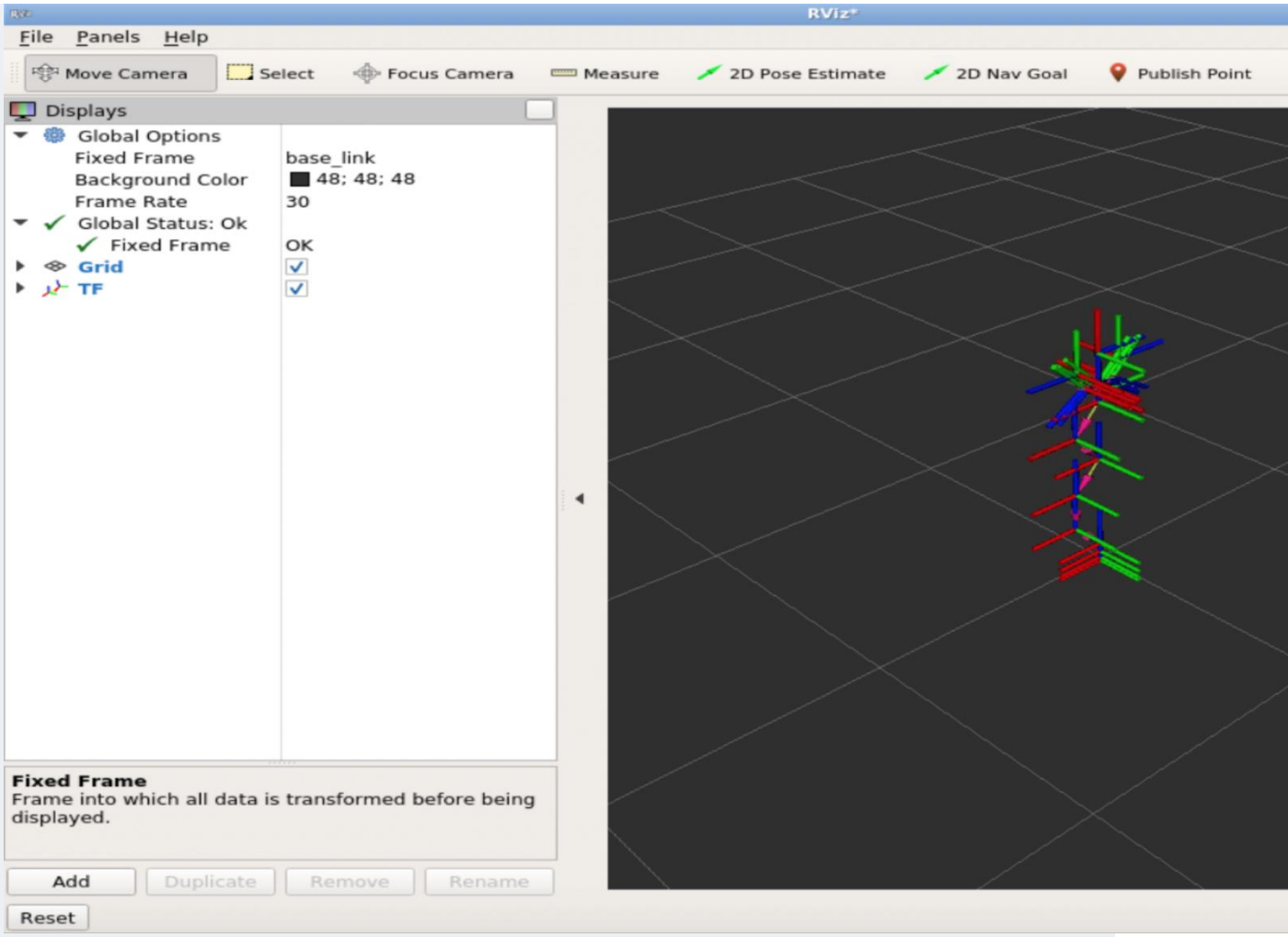
- The *Add button*. Clicking here will give you all of the types of elements that can be represented in RVIZ.

Go to RViz2 in the graphical interface and add a TF element. For that, click "Add" and select the element TF in the list of elements provided, as shown in the image below.



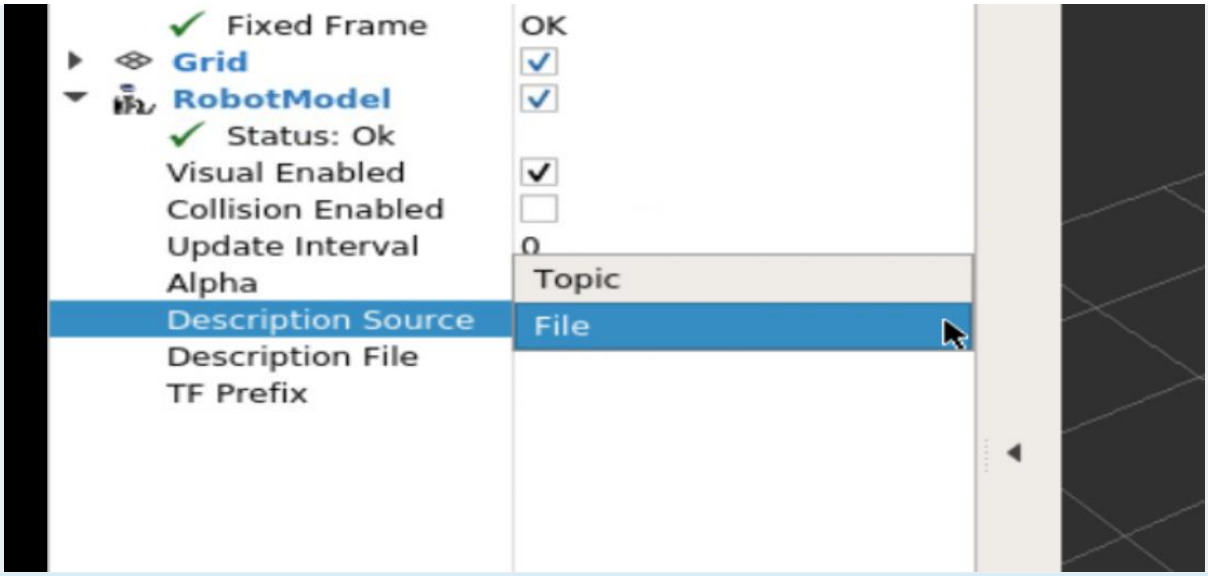
RViz2 Add element

- Go to the RViz2 left panel, select the *base_link* as Fixed Frame, and make sure that the TF element checkbox is checked. In a few moments, you should see all of the Robot's Elements Axis represented in the CENTRAL Panel.



RViz2 TF's

- Now, press "Add" and select *RobotModel*.
- On the RobotModel options, set the "Description Source" to **File**.



- Notes -

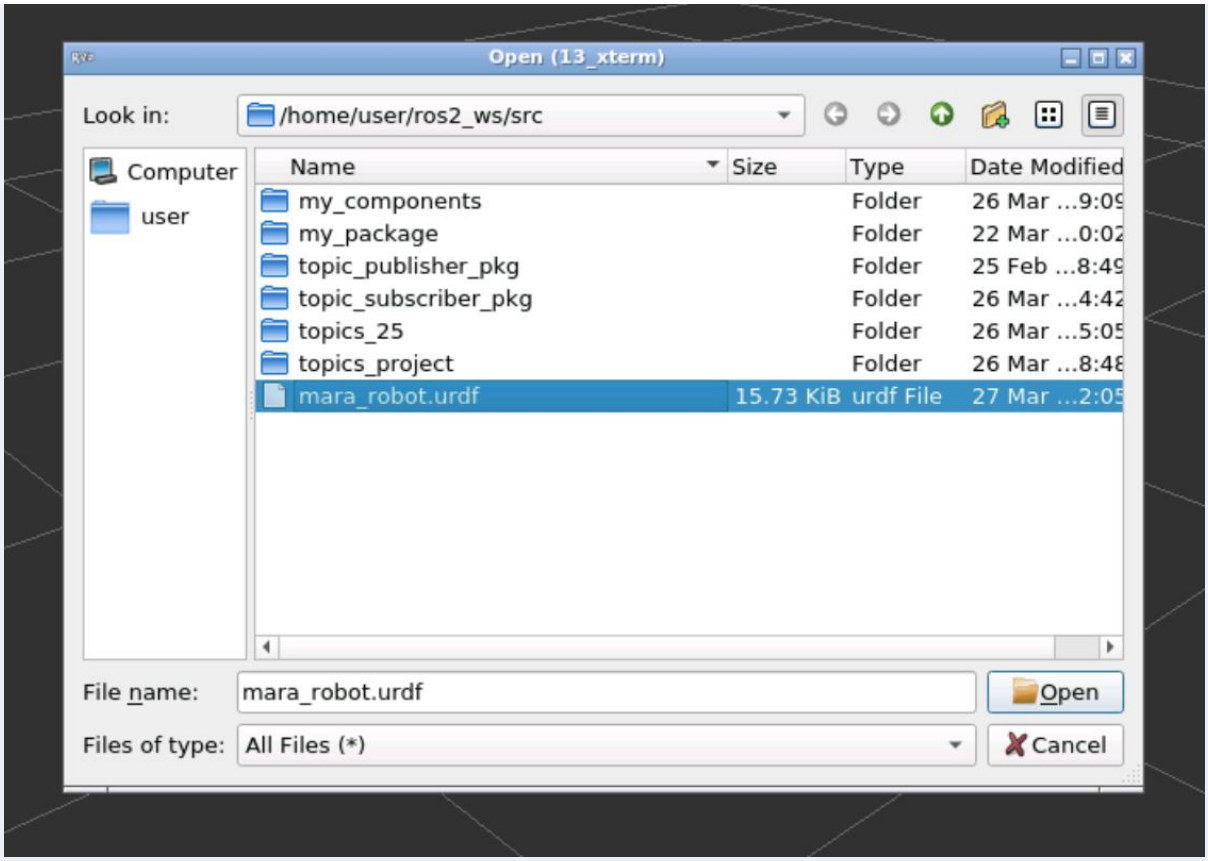
In order to advance in the chapter, you need to copy the URDF file of the robot into your workspace. You can this by exdecuting the following commands:

Execute in Shell #2

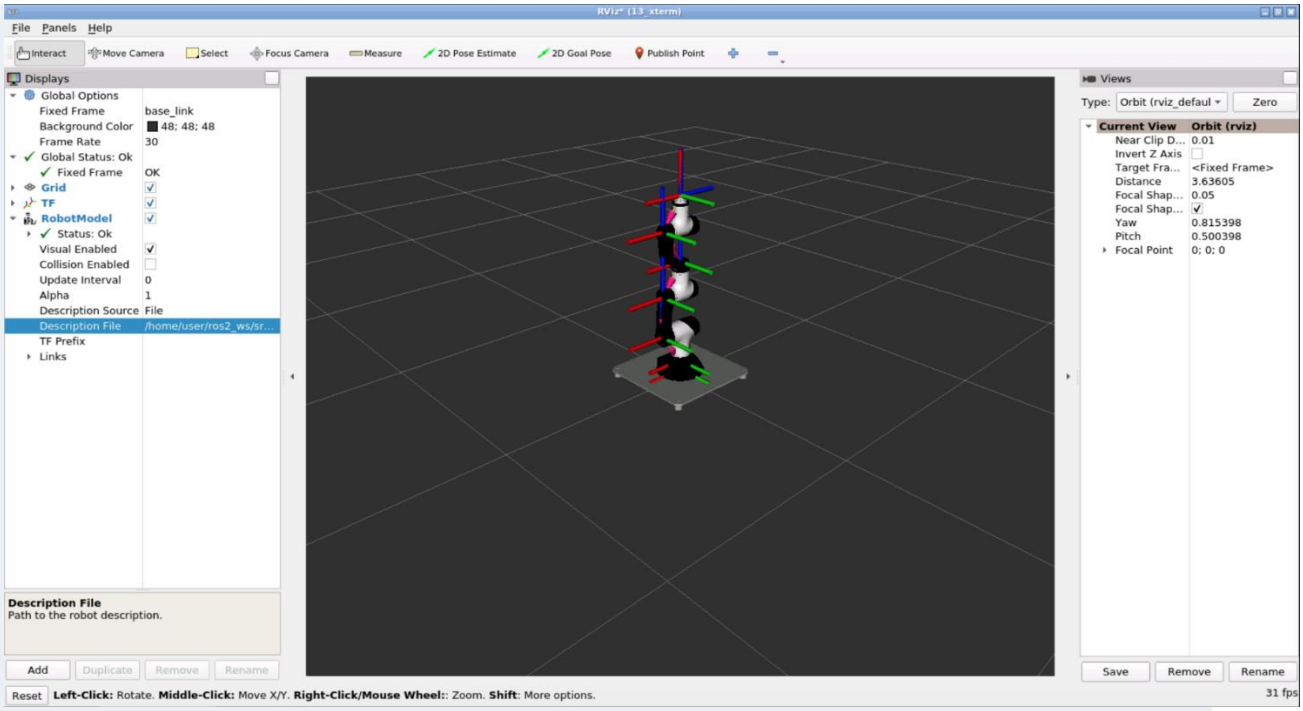
```
$ cd /home/simulations/ros2_sims_ws/src/ros2_mara/mara/mara_description/urdf
$ cp mara_robot.urdf ~/ros2_ws/src/
```

- End of Notes -

- Now, in the "Description Source" option, select the file named **mara_robot.urdf**.



You should now see the 3D model of the robot, as shown in the image below:



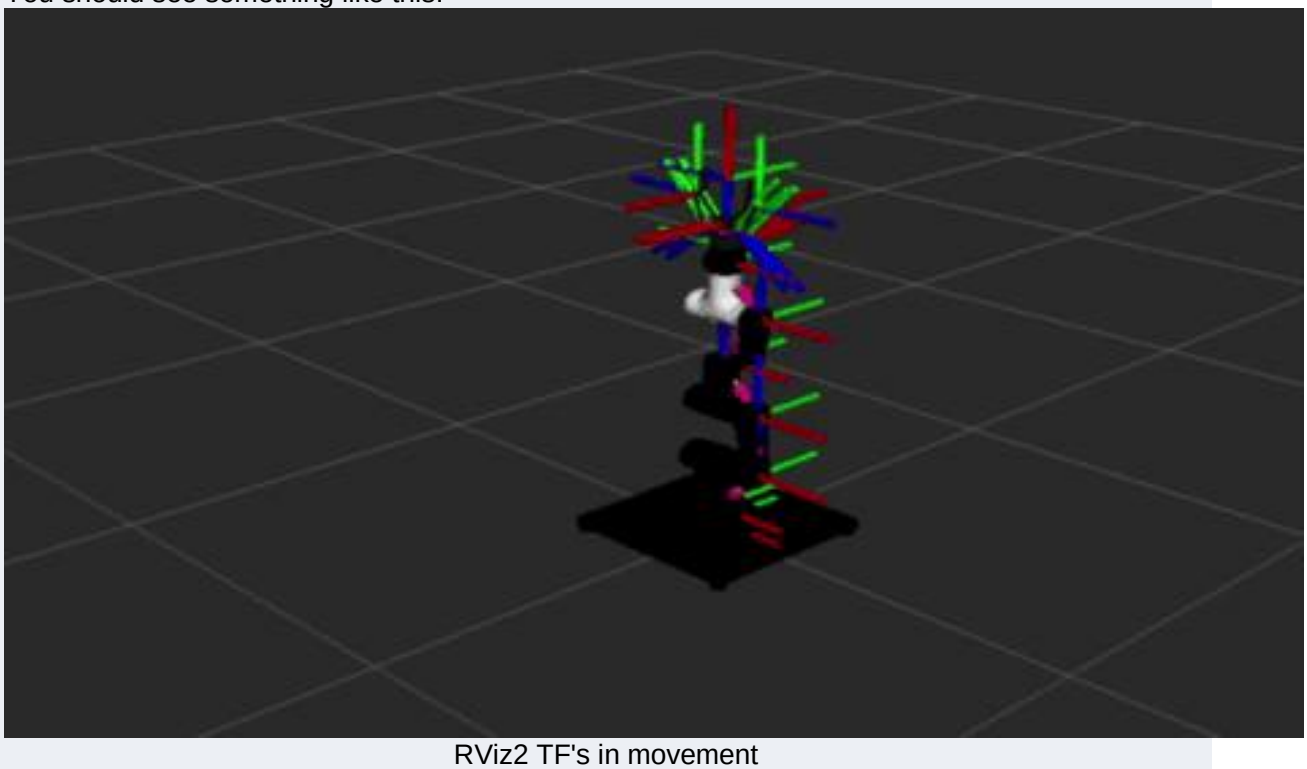
RViz2 Robot Model

- Now, go to WebShell #2 and enter the command to move the robot:

Execute in Shell #2

```
$ source .bashrc_ros2
$ ros2 run mara_minimal_publisher mara_minimal_publisher_v1.py
```

You should see something like this:



RViz2 TF's in movement

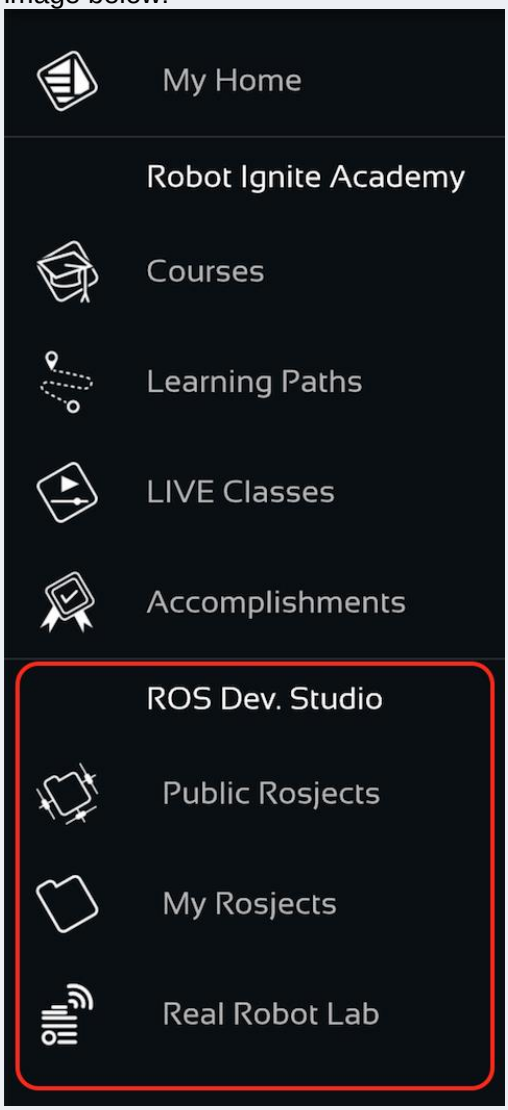
In the above GIF, you are seeing all of the transformation elements of the Mara robot in real-time. This allows you to see exactly what joint transformations are sent to the robot arm to check if it's working properly.

Final Recommendations ¶

Now that you have finished the course, you may be wondering... **what should I do now?** Well, let us provide you with some options that you can take in order to keep pushing your ROS learning!

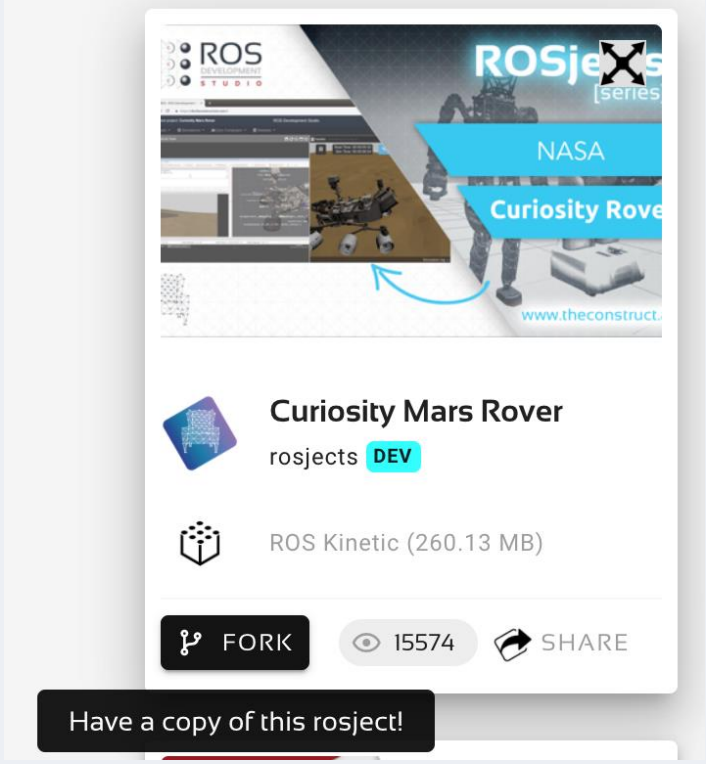
Option 1: ROS Development Studio (ROSDS)

ROSDS is the The Construct web based tool to program ROS robots online. It requires no installation in your computer. Hence, you can use any type of computer to work on it (Windows, Linux or Mac). You can access ROSDS in the main page's left menu, as you can see in the image below:

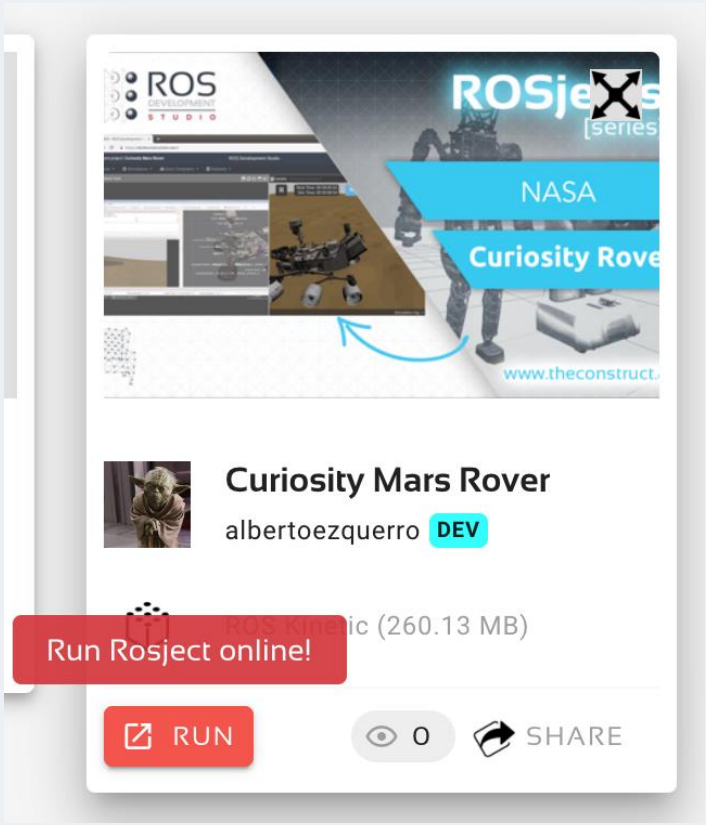


As you can see, ROSDS is divided into 3 main sections: **Public Rosjects**, **My Rosjects** and **Real Robot Lab**.

In the **Public Rosjects** section you will find many available public rosjects created by the ROS Developers community. You can use any of this rosjects in order to apply what you've learned during this course. To use a public rosject, you will need to **fork** it. You can fork an specific rosject by cliclking on its Fork button.



Once you fork a rosject, it will appear in the **My Rosjects** section. This is the section where you will be able to manage all your rosjects. Here, you will be able to start the rosject by pressing on its Run button.



Below you will find a list of rosjects that are interesting for practicing the contents of this course:

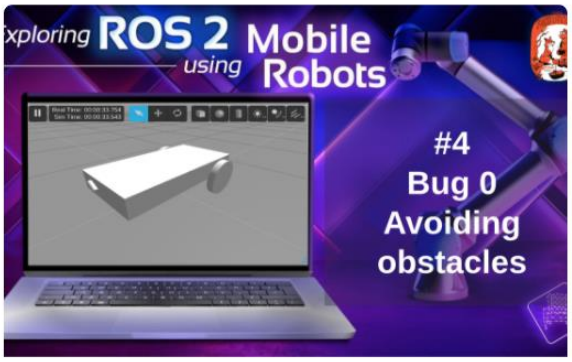
Rosject 1

ROS Developers Live Class n.110: Generate topic statistics with ROS2

rosjects DEV

ROS2 Foxy (464.23 MB)

Rosject 2

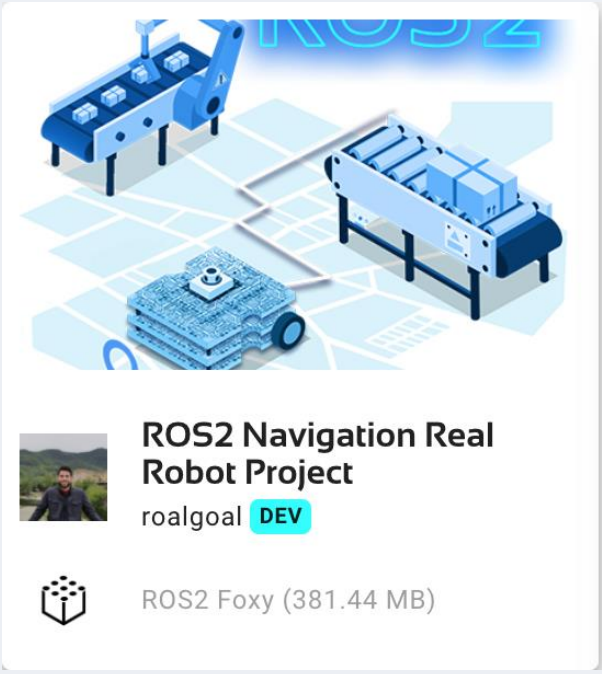


Exploring ROS2 using wheeled Robot - #2 - Reading sensor data

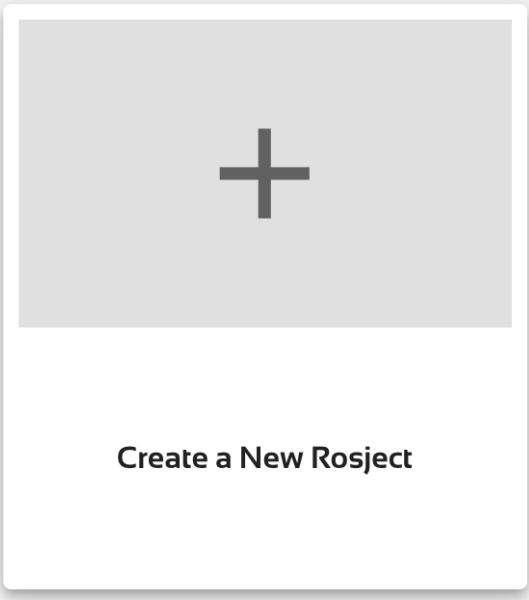
marco.nc.arruda

ROS2 Foxy (159.26 MB)

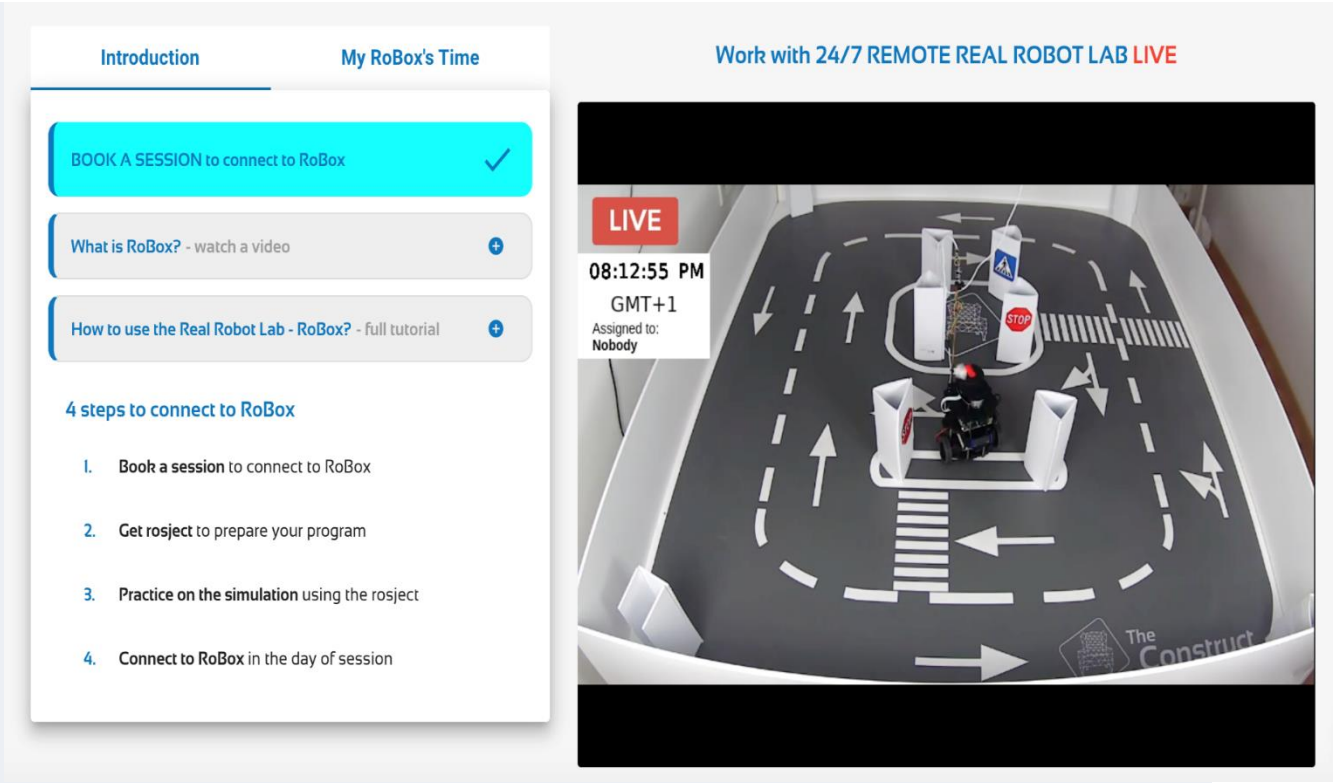
Rosject 3



if you feel confident enoguh, in the **My Rosjects** section you also have the option to create your own rosject!

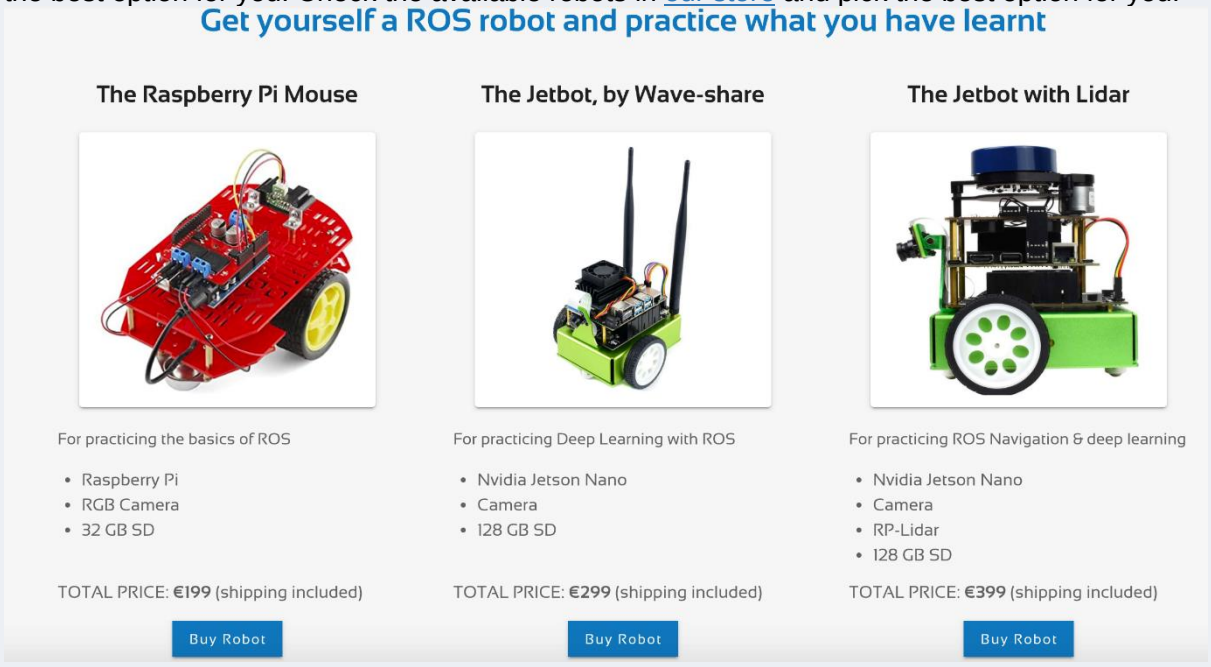


Finally, in the **Real Robot Lab** section you will also be able to practice with a real robot! Book a session, create your ROS programs and test them on a real robot based in our lab in Barcelona!



Option 2: Buy a robot

Would you like to start applying what you've learned in a real robot of your own? Then this might the best option for you! Check the available robots in our store and pick the best option for you!



Option 3: Keep learning!

Of course, once you have finished this course, you can still learn a lot of other interesting ROS subjects.

For instance, you could add to your current skills ROS2 Navigation, in order to make your robots capable of navigate autonomously.

Or check any of the other many courses we have in the Academy!

Hope to see you soon!