ROS2 BASICS IN 5 DAYS

Ricardo Téllez

Alberto Ezquerro

Miguel Angel Rodríguez

The Construct Sim, Gran Vía 608, 3-D 08007 Barcelona SPAIN,

+34 687 672 123, info@theconstructsim.com www.theconstruct.ai

Written by Miguel Angel Rodríguez, Alberto Ezquerro and Ricardo Téllez
Edited by Yuhong Lin and Ricardo Téllez
Cover design by Lorena Guevara
Learning platform implementation by Ruben Alves

Version 1.0

The Construct Sim, Gran Vía 608, 3-D 08007 Barcelona SPAIN,
+34 687 672 123, info@theconstructsim.com www.theconstruct.ai

Index

Preface

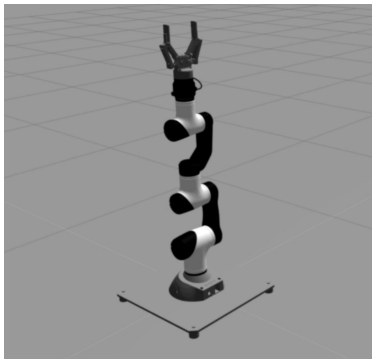
Preface

This book relies on a series of Gazebo simulations in order to teach you ROS2. You need to have the simulations running while following the course, if you really want to learn ROS2 in 5 days, since we heavily rely on them to explain you the complex concepts of ROS.

The simulations

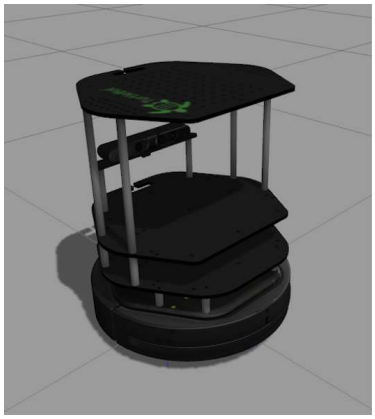
The list of simulations that you need to execute are the following:

MARA



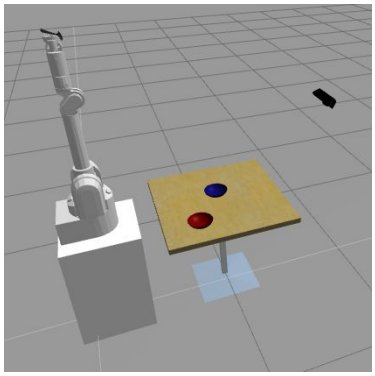
MARA

Repository: https://bitbucket.org/theconstructcore/ros2_mara/src/master/
Kobuki



Kobuki

Repository: https://bitbucket.org/theconstructcore/turtlebot/src/master/
WAM



Wam Arm

Repository: https://bitbucket.org/theconstructcore/iri_wam/src/master/
BB-8

BB8

Repository: https://bitbucket.org/theconstructcore/bb8/src/master/

How to launch the simulations

In order to launch the simulations you have two options:

1- **Launch the simulations in our ROS Development Studio (ROSDS) using the ROSjects we provide**. ROSDS is the The Construct web based tool to program ROS robots online. It is recommended that you use this option since it is very simple and requires no installation in your computer. Hence, you can use any type of computer to follow this book (Windows, Linux or Mac). Additionally, free accounts are available. You just need to paste the ROSject link we will provide to you in each Unit to a browser's URL, and you will automatically have the simulation prepared in your ROSDS workspace.

2- **Launch the simulations in your own computer**. This option requires that you have a running Linux computer with ROS2 and Gazebo simulator installed in your computer. We do not cover those steps (but more information below)

1. Launch the simulation using ROSjects

You will find a complete guide about **ROSjects** and how to open them in the next Chapter of the Book.

2. Launch the simulation in your own computer

We do not recommend this option because it requires you to know about ROS concepts that you still don't have (because you are going to learn them in this book). However, we provide this option for more advanced users who more or less know what they are doing. If that is not your case, use the option of launching the simulations on the ROSDS, which requires no previous knowledge of ROS.

For this option you will need to:

1- Download the simulations from the repository, following the links provided above.

2- Install and set up a ROS2/ROS1 environment in your PC. The instructions to do that are explained in the official page of ROS (www.ros.org). Bear in mind that the simulations you downloaded are supported for ROS2 Crystal / ROS Melodic with Gazebo 9. Any other setup might not compile or work correctly.

Consultations

If you have any doubts while doing the Course, or you get really sucked in some of the exercises, you can ask for support in our forum:

http://forum.theconstructsim.com

ROSjects

ROSjects

Throughout the whole book, you're going to **find a ROSject at the beginning of each Chapter**. With these ROSjects, you are going to be able to easily have access to all the material you'll need for each Chapter.

What is a ROSject?

A ROSject is, basically, a ROS project in the ROS Development Studio (ROSDS). ROSjects can easily be shared using a link. By clicking on the link, or copying it to the URL of your web browser, you will have a copy of the specific ROSject in your ROSDS workspace. This means you will have instant access to the ROSject. Also, you will be able to modify it as you wish.

How to open a ROSject?

At the beginning of each Chapter, you will see a section like this one:

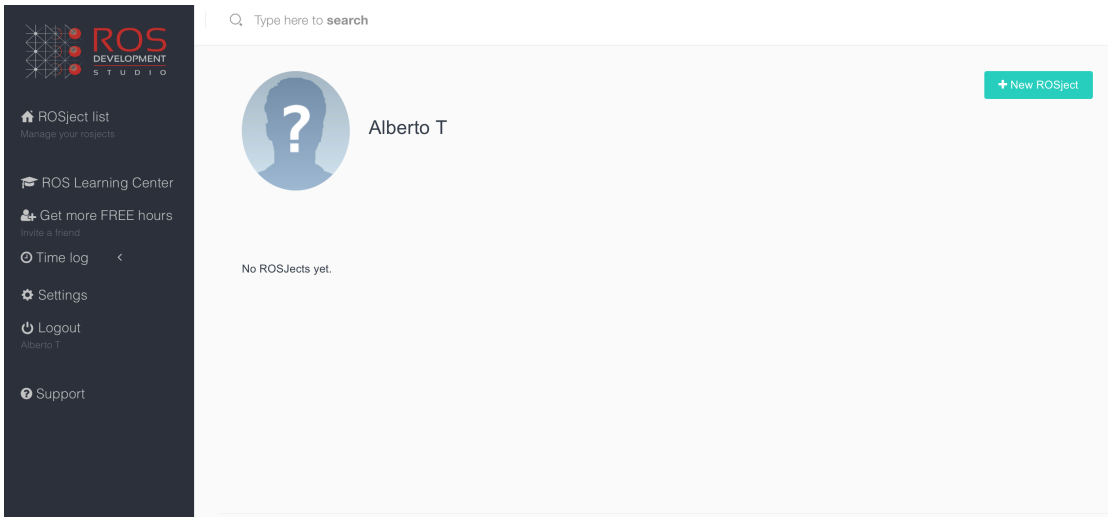
ROSject section

As you can see, it contains 3 things:

- **ROSject Link:** Link to get the ROSject
- **Robot:** Name of the robot to launch in the Simulations list at ROSDS.

Step 1

Log into the ROSDS platform at http://rosds.online . If you don't have an account, you can create one for free. Once you log in, you will see an screen like the below one.

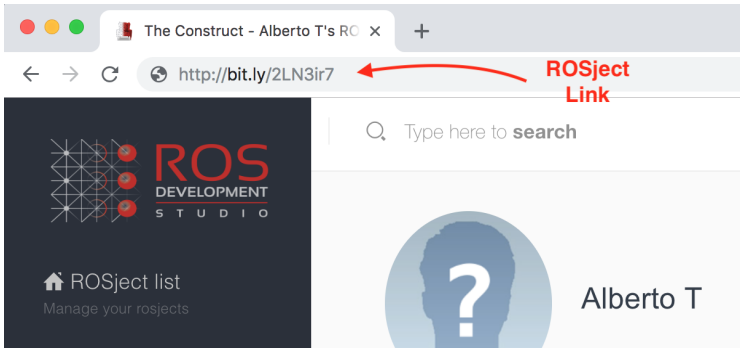


ROSjects Empty List

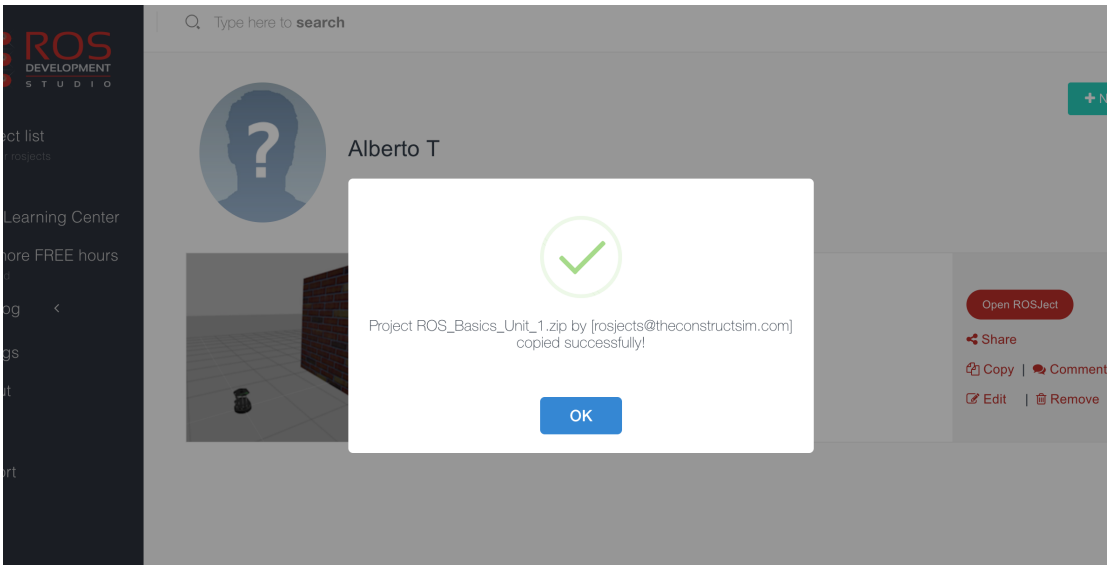
At this point you don't have any ROSject yet. So let's get one!

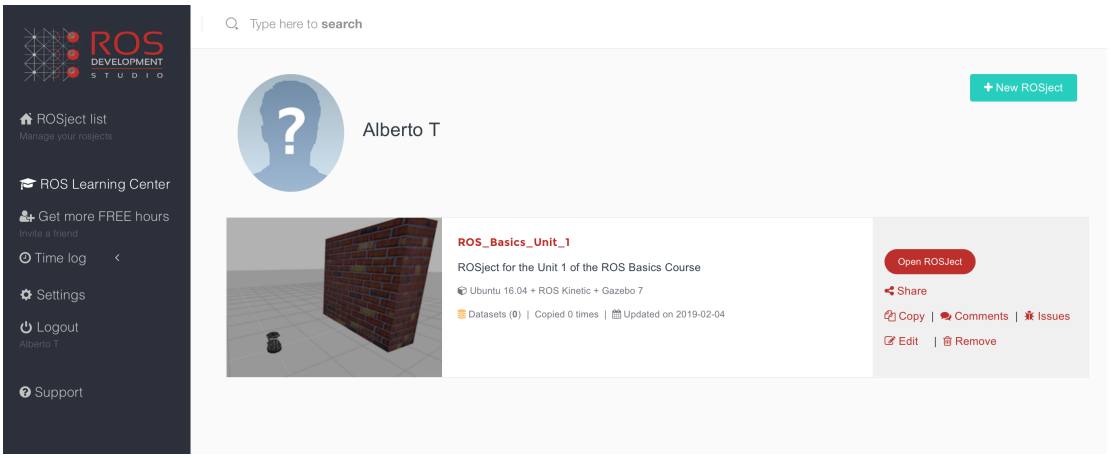
Step 2

Copy the ROSject Link to your web browser. Once you have copied the URL to your web browser, you will automatically have that ROSject available in your workspace.



ROSject Link

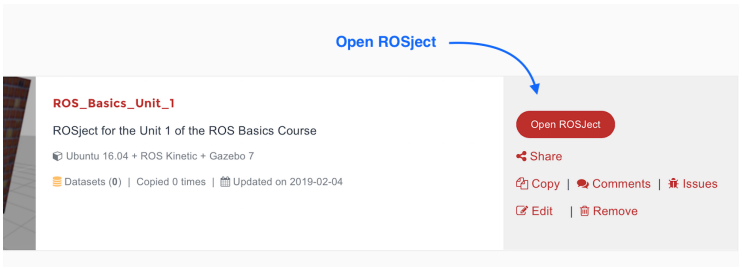




ROSject in ROSDS workspace

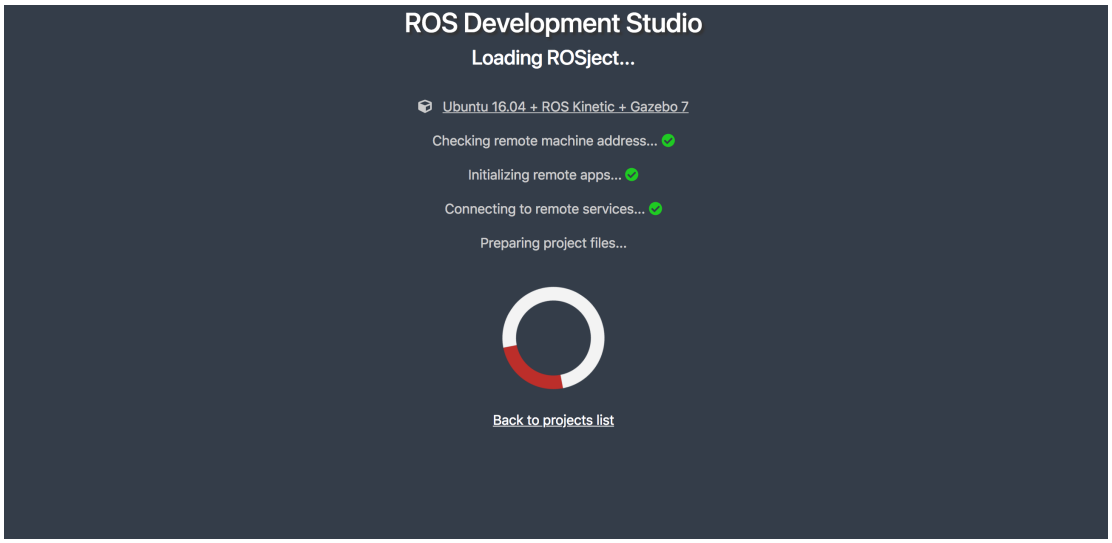
Step 3

Open the ROSject. You can open the ROSject by clicking on the **Open ROSject** button.

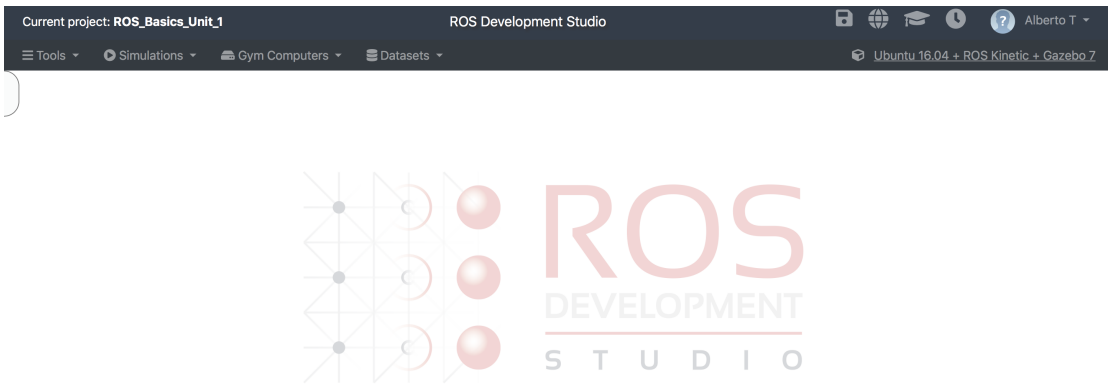


Open ROSject

You will then go to a loading screen like the below one.



After a few seconds, you will get an environment like the below one.



ROSDS Environment

Contents of the ROSjects

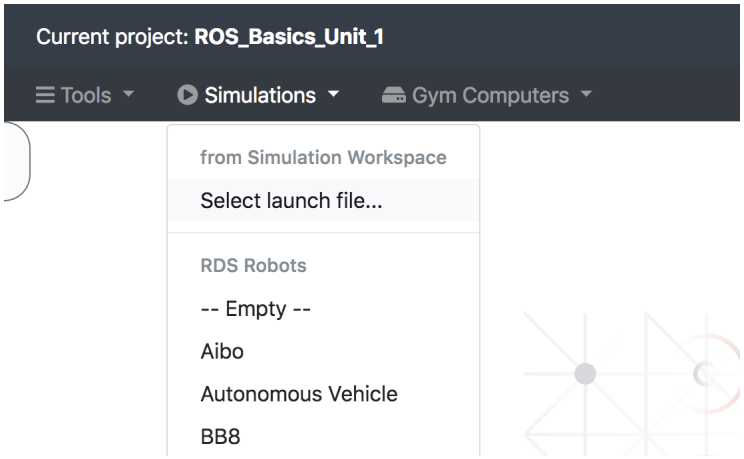
The ROSjects that are available for each chapter of the book will basically contain 2 things:

- The Gazebo simulation used for the Chapter
- All the scripts and files used for the Chapter

In order to open the simulation, follow the next steps:

Step 1

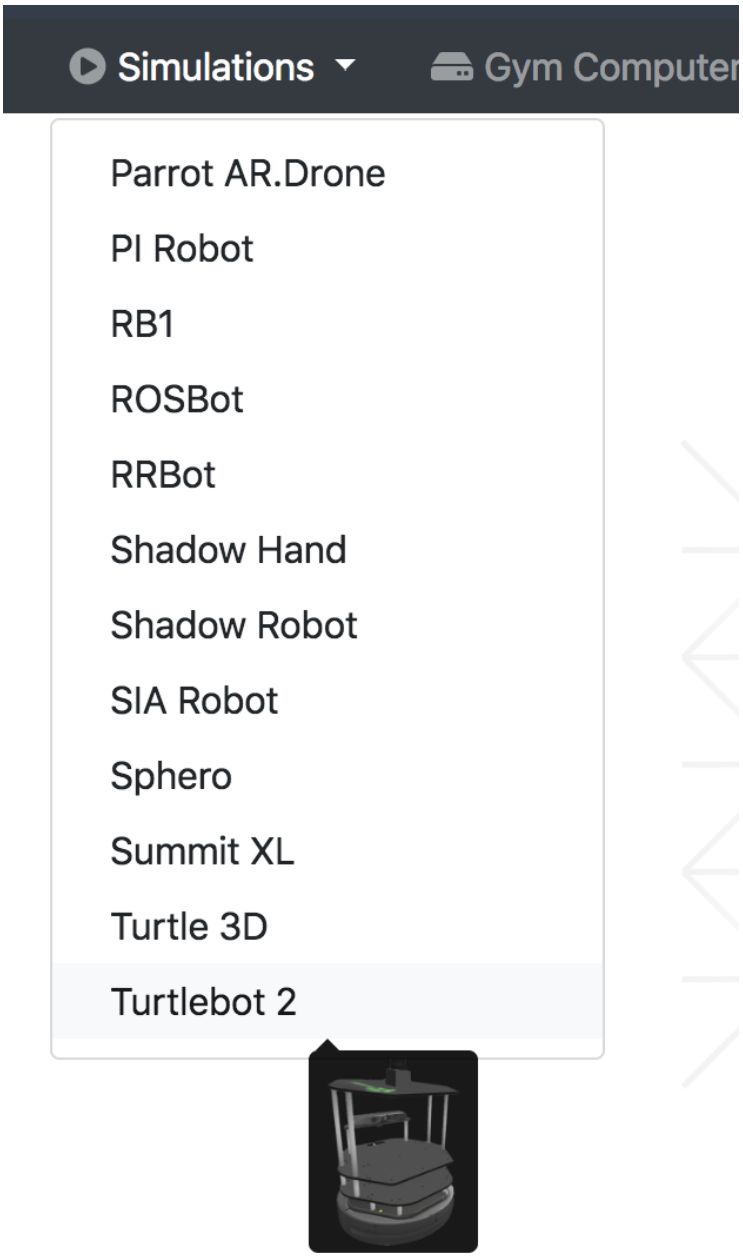
In the **Simulations** menu, you will get a list of all the robots you can launch under **RDS Robots**.



Simulations menu

Step 2

Within the list, select the **Robot** specified at the ROSject section of the chapter.

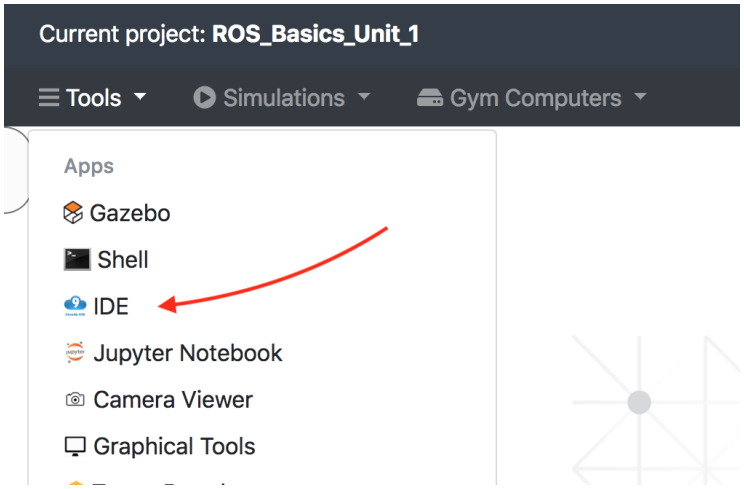


RDS Robots list

In order to see all the files related to the chapter, follow the next steps:

Step 1

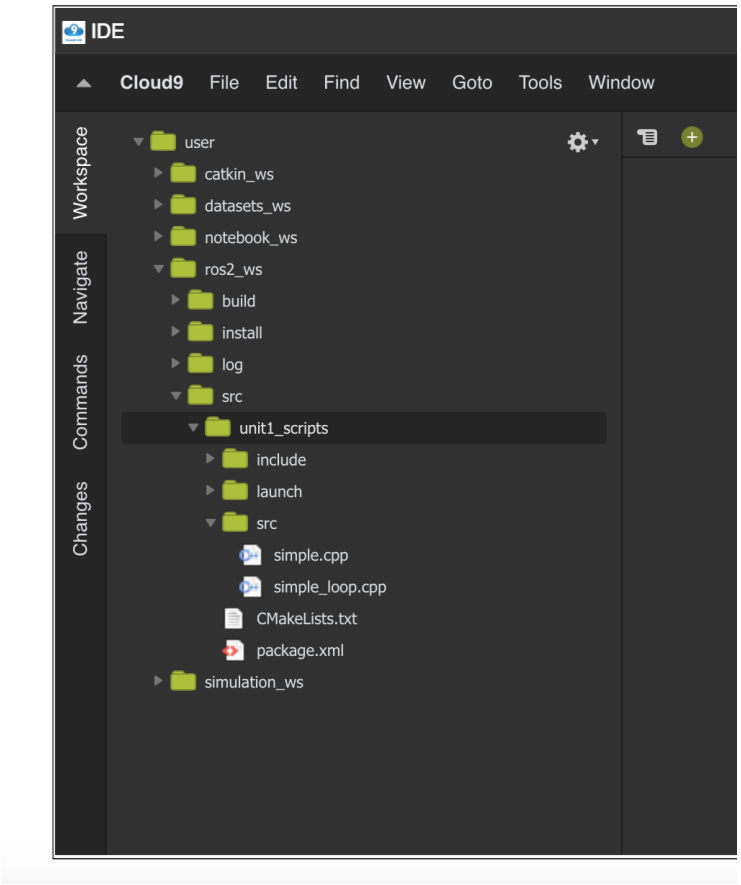
In the **Tools** menu, click on the **IDE** option. An IDE will appear in your workspace.



Tools menu

Step 2

You will find all the files inside the **ros2_ws** workspace. There, you have a ROS2 package containing all the files related to the chapter. This package will always follow the following name convention: **<unitX>_scripts**



IDE workspace tree

And that's it! Now just enjoy ROSDS and push your ROS learning.

Also, you can find more information and videotutorials about ROSDS here: https://www.youtube.com/watch?v=ELfRmuqgxns&list=PLK0b4e05LnzYGvX6EJN1gOQEl6aa3uyKS If you have any doubt regarding ROSDS, don't hesitate in contacting us to: **feedback@theconstructsim.com**

Unit 0. Introduction to the Course

ROS2 BASICS IN 5 DAYS

Unit 0: Introduction to the Course

- ROSject Link: https://bit.ly/2TfBPxU
- Robot: **MARA**

NOTE: You will find instructions on how to launch this simulation in the Jupyter Notebook of the ROSject.

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.

What's this course about?

Since ROS started back in 2007, a lot has changed in the robotics world and, with it, in the ROS community. What started as a "small" project has become the main tool for robot developers all around the world. This means that ROS is being pushed to its limits every day. With all this in mind, and in order to accomplish all the new challenges that robotics evolution is presenting, ROS is now ready to evolve. And this evolution is none other than ROS2.

The goal of ROS2 is to bring ROS to a whole new level, maintaining all the awesome features that ROS already provides, and adding many new functionalities that will make sure that ROS2 can fulfill all the new challenges that robotics will bring in the years to come.

So, the goal of this course will be 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?

Do you want to have a taste?

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 1.1

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

Execute in WebShell #1

```
[ ]: source /opt/ros/crystal/setup.bash
```

```
[ ]: source /home/simulations/ros2_sims_ws/install/setup.bash
```

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

Execute in WebShell #1

```
[ ]: ros2 run mara_minimal_publisher mara_minimal_publisher
```

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 down. 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!

What will you learn with this course?

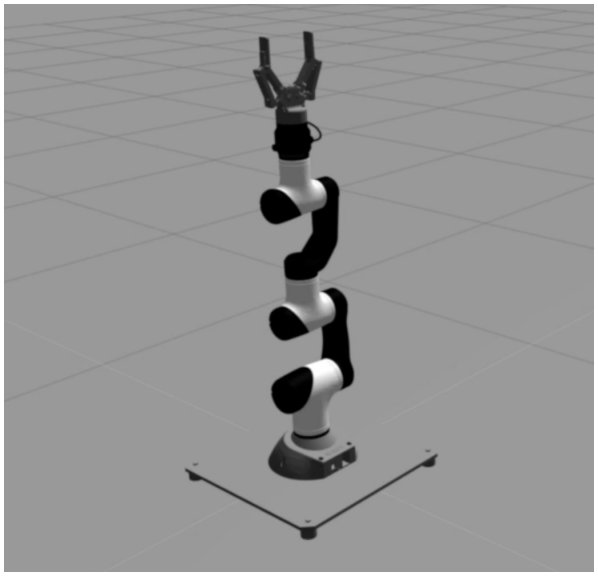
Basically, during this course, you will address the following topics:

- Basic Concepts of ROS2: Packages, Launch Files, Nodes, Client Libraries, etc...
- How to work with ROS1 Bridge
- How Topics work: Publishers and Subscribers
- How Services work: Clients and Servers
- Basic Debugging Tools: Logging system, RViz2.

How will you learn all this?

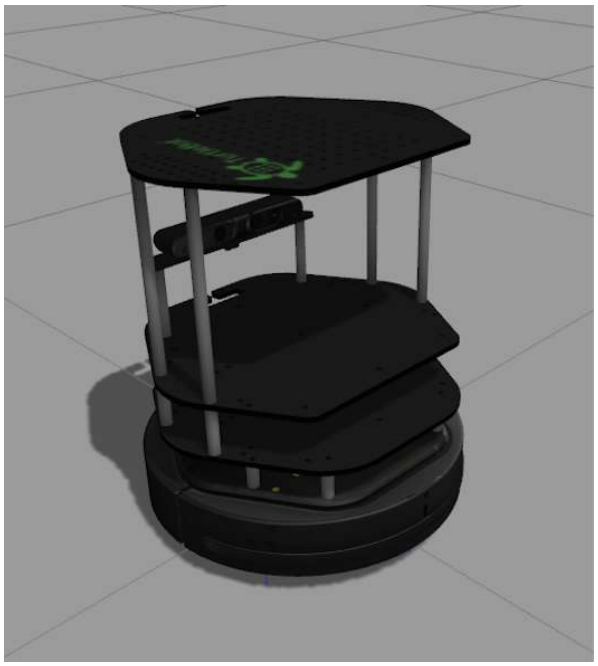
You will learn through hands-on experience from day one! During the course, you will work with the following simulations.

MARA robot:



MARA Robot

Turtlebot 2:

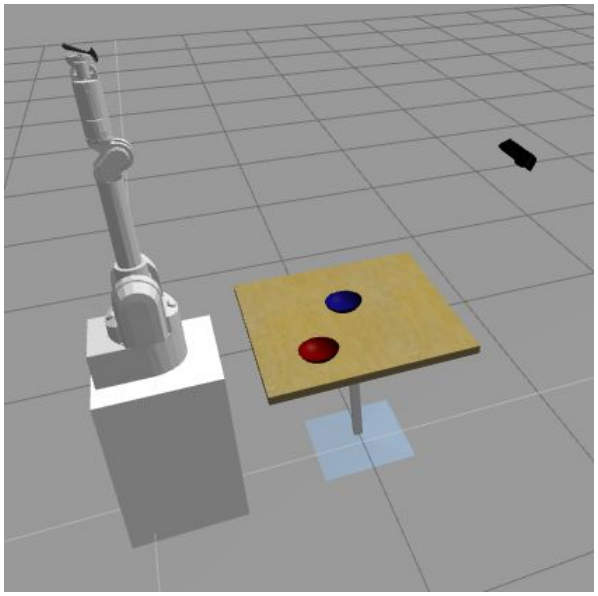


Turtlebot 2

BB8:

BB-8

Wam Arm:



WAM Arm

Minimum requirements for the course

In order to be able to fully understand the contents of this course, it is highly recommended that you have the following knowledge:

- Basic C++.
- Basic Unix shell knowledge.

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

Erle Robotics

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

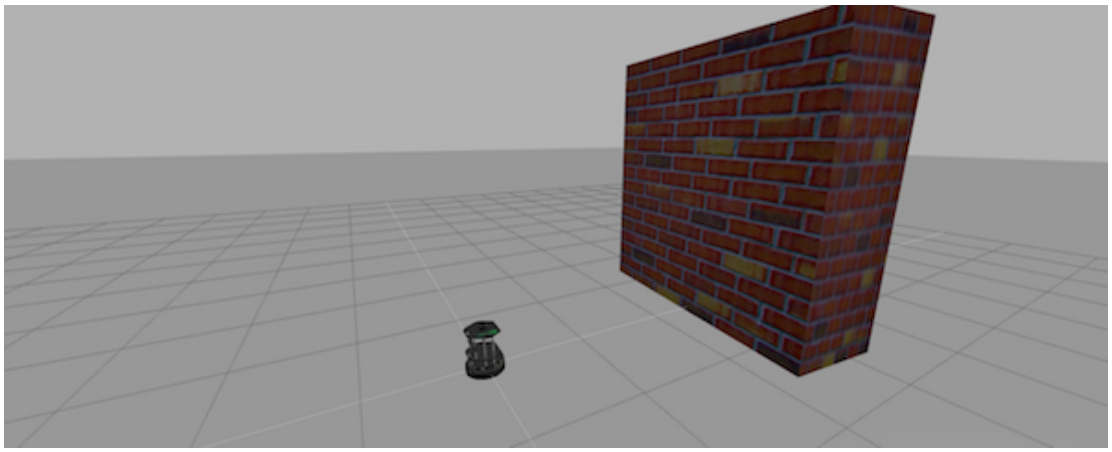
ROS

Gazebo

Unit 1. Basic Concepts

ROS2 BASICS IN 5 DAYS

Unit 1: Basic Concepts



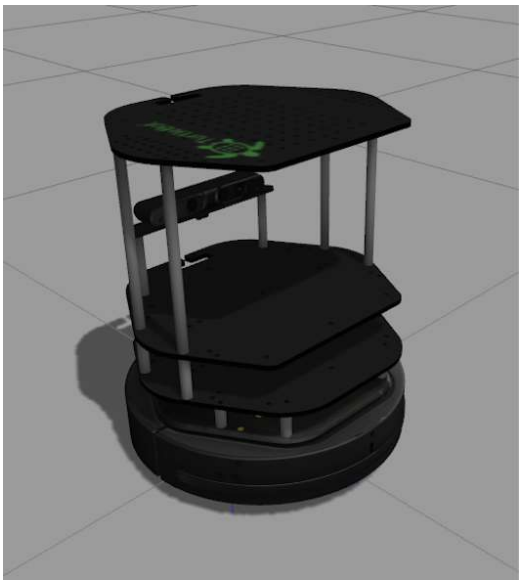
Kobuki

Run on ROSDS

- ROSject Link: https://bit.ly/2CJRBtT
- Robot: **Turtlebot 2**

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.



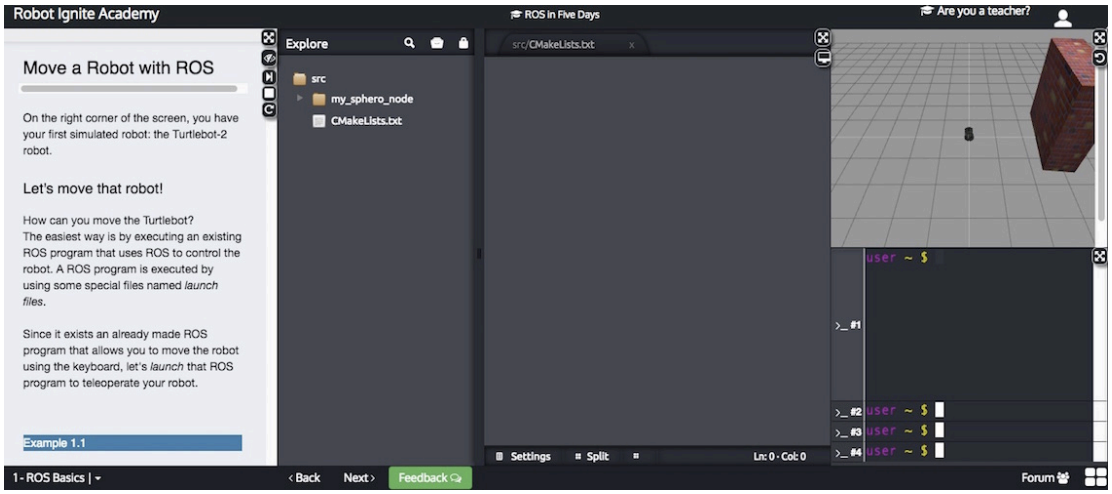
Kobuki Robot

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.

Move a Robot with ROS2

On the right corner of the screen, you have your first simulated robot: the Turtlebot 3 robot against a large wall.



Robot Ignite Environment

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 that allows you to move the robot using the keyboard, let's launch that ROS2 program to teleoperate the robot.

Example 1.1
Execute the following commands in WebShell #1 in order to start the **ROS1 Bridge**.

Execute in WebShell #1

```
[ ]: source ~/.bashrc_bridge

[ ]: export ROS_MASTER_URI=http://localhost:11311

[ ]: ros2 run ros1_bridge dynamic_bridge
```

Now, execute the following commands in WebShell #2
Execute in WebShell #2

```
[ ]: source /opt/ros/crystal/setup.bash

[ ]: ros2 run teleop_twist_keyboard teleop_twist_keyboard
```

WebShell #1 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
```

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

Keys to control Kobuki Robot

Try it!! When you're done, you can **Ctrl+C** to stop the execution of the program.

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).

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, this package 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.

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.

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 1.2
In **Example 1.1**, 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

```
[ ]: """Launch a talker and a listener."""

     from launch import LaunchDescription
     import launch_ros.actions


     def generate_launch_description():
         return LaunchDescription([
             launch_ros.actions.Node(
                 package='teleop_twist_keyboard',
     node_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',
     node_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. **node_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

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 1.3
First of all, let's source ROS2 on our Shell, so that we can use the ROS2 command line tools.

Execute in WebShell #1

```
[ ]: source /opt/ros/crystal/setup.bash
```

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

Execute in WebShell #1

```
[ ]: cd ~/ros2_ws/
     pwd
```

WebShell #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 WebShell #1

```
[ ]: cd src
```

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

Execute in WebShell #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>
```

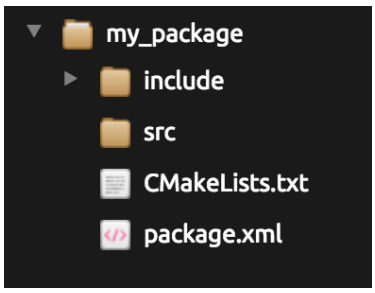
The **package_name** is the name of the package you want to create, and the **package_dependencies** are the names of other ROS packages that your package depends on.

Example 1.4
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 WebShell #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 1.1})
Fig.1.1 - IDE created package my_package



IDE created 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.

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 --symlink-install
```

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 1.5
Go to your **ros2_ws** directory and compile your source folder. You can do this by typing:

Execute in WebShell #1

```
[ ]: cd ~/ros2_ws
     colcon build --symlink-install
```

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 --symlink-install --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 WebShell #1

```
[ ]: colcon build --symlink-install --packages-select my_package
```

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

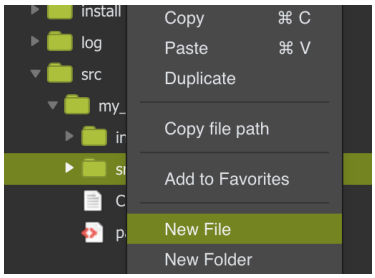
Execute in WebShell #1

```
[ ]: source ~/ros2_ws/install/setup.bash
```

My first ROS program

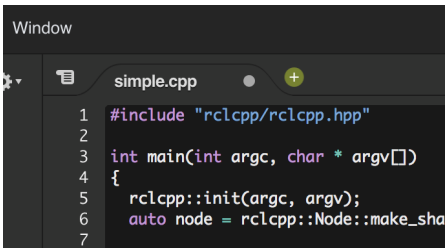
At this point, you should have your first package created... but now, you need to do something with it! Let's do our first ROS2 program!

Example 1.6
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,.

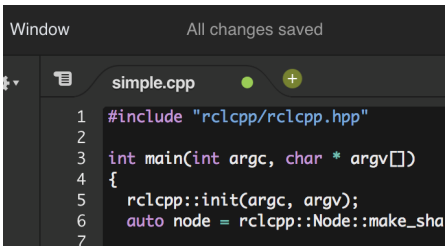


New File Creation

A new Tab should have appeared on the IDE with empty content. Copy the content of simple.cpp into the new file. Finally, press **Ctrl-S** to save your file with the changes. The Gray Dot in the Tab will go from Gray to Green (see pictures below). Also a message will appear saying All changes saved.



Unsaved



Saved

2- Create a launch directory inside the package named my_package {Example 1.4}.

Execute in WebShell #1

```
[ ]: cd ~/ros2_ws/src/my_package
     mkdir launch
```

You can also create it through the IDE.

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

Execute in WebShell #1

```
[ ]: touch launch/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 1.3}.

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}/
     )
```

```
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
```

CMakeLists.txt

6- Compile your package as explained previously.

Execute in WebShell #1

```
[ ]: cd ~/ros2_ws;
     colcon build --symlink-install
     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]
```

colcon build output

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

Execute in WebShell #1

```
[ ]: ros2 launch my_package my_package_launch_file.launch.py
```

Expected Result for Example 1.6
You should see Leia's quote among the output of the roslaunch command.

WebShell #1 Output

```
[ ]: user:~/ros2_ws$ ros2 launch my_package
     my_package_launch_file.launch.py
     [INFO] [launch]: process[simple_node-1]: started with pid [29595]
     [INFO] [ObiWan]: Help me Obi-Wan Kenobi, you're my only hope
     [INFO] [launch]: process[simple_node-1]: process has finished cleanly
```

C++ Program {1.1a-cpp}: 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**
Launch File {1.1-l}: 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 a talker and a listener."""

     from launch import LaunchDescription
     import launch_ros.actions


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

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.

ROS 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 1.7
Type this command in a new shell and look for the node you've just initiated (ObiWan).

Execute in WebShell #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 {1.1b-cpp}: 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
```

Launch your program again using the roslaunch command.

Execute in WebShell #1

```
[ ]: ros2 launch my_package my_package_launch_file.launch.py
```

Now, try again in another WebShell:

Execute in WebShell #2

```
[ ]: source /opt/ros/crystal/setup.bash
```

```
[ ]: ros2 node list
```

Can you see your node now?

WebShell #1 Output

```
[ ]: user:~$ ros2 node list
     /launch_ros
     /ObiWan
```

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

```
[ ]: ros2 node info /ObiWan
```

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

Execute in WebShell #1

```
[ ]: ros2 node info /ObiWan
```

WebShell #1 Output

```
[ ]: user:~$ ros2 node info /ObiWan
     /ObiWan
       Subscribers:
         /parameter_events: rcl_interfaces/ParameterEvent
       Publishers:
         /parameter_events: rcl_interfaces/ParameterEvent
       Services:
         /ObiWan/describe_parameters: rcl_interfaces/DescribeParameters
         /ObiWan/get_parameter_types: rcl_interfaces/GetParameterTypes
         /ObiWan/get_parameters: rcl_interfaces/GetParameters
         /ObiWan/list_parameters: rcl_interfaces/ListParameters
         /ObiWan/set_parameters: rcl_interfaces/SetParameters
         /ObiWan/set_parameters_atomically:
     rcl_interfaces/SetParametersAtomically
```

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

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/

Environment Variables

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

```
[ ]: export | grep ROS
```

NOTE 1: Depending on your computer, it could happen that you can't type the | symbol directly in your webshell. If that's the case, just **copy**/**paste** the command by **RIGHT-CLICKING** on the WebShell and select **Paste from Browser**. This feature will allow you to write anything on your webshell, no matter what your computer configuration is.

```
[ ]: user:~$ export | grep ROS
     declare -x ROS_DISTRO="crystal"
     declare -x ROS_IP="10.8.0.1"
     declare -x ROS_MASTER_URI="http://10.8.0.1:11311"
     declare -x ROS_PYTHON_VERSION="3"
     declare -x ROS_VERSION="2"
```

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. Usually, your own computer (localhost).
     ROS_DISTRO -> Contains the ROS distribution that you are currently
     using.
```

NOTE: 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.

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!

Unit 2. ROS1 Bridge

ROS2 BASICS IN 5 DAYS

- ROSject Link: https://bit.ly/2B5pnK4
- Robot: **BB-8**

Unit 2: ROS1 Bridge

Estimated time to completion: 6 hours What will you learn with this unit?

- Setup of ROS1 Bridge
- Basic Examples
- Custom Message Mapping
- Exercises with simulation

Setup of ROS1 Bridge

At the time of the creation of this course notebook, ROS2 Crystal 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.

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 WebShell #1

```
[ ]: # Check a Custom bahsrc_bridge
     cat /home/user/.bashrc_bridge
```

Setup {4.1}: .bashrc_bridge

```
[ ]: source /opt/ros/melodic/setup.bash
     source /opt/ros/crystal/local_setup.bash

     source /home/user/catkin_ws/devel/setup.bash
     source /home/user/ros2_ws/install/local_setup.bash
```

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 **crystal** 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

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 {4.1}: .bashrc_ros1

```
[ ]: # ROS1
     export ROS_DISTRO=melodic
     source /opt/ros/$ROS_DISTRO/setup.bash
     source /home/user/catkin_ws/devel/setup.bash
```

Setup {4.1}: .bashrc_ros2

```
[ ]: # ROS2
     export ROS_DISTRO=crystal
     source /opt/ros/$ROS_DISTRO/setup.bash
     source /home/user/ros2_ws/install/local_setup.bash
```

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 WebShell #1: ROS1-Bridge
We launch ROS1-Bridge, sourcing for both ROS1 and ROS2, using the provided **.bashrc_bridge**.

```
[ ]: . /home/user/.bashrc_bridge
     export ROS_MASTER_URI=http://localhost:11311
     ros2 run ros1_bridge dynamic_bridge
```

Execute in WebShell #2: Talker ROS1

```
[ ]: . /home/user/.bashrc_ros1
     rosrun rospy_tutorials talker
```

Execute in WebShell #3: Listener ROS2

```
[ ]: . /home/user/.bashrc_ros2
     ros2 run demo_nodes_cpp listener
```

WebShell #1 Output: ROS1-Bridge

```
[ ]: created 1to2 bridge for topic '/chatter' with ROS 1 type
     'std_msgs/String' and ROS 2 type 'std_msgs/String'
     [INFO] [ros1_bridge]: Passing message from ROS 1 std_msgs/String to
     ROS 2 std_msgs/String (showing msg only once per type)
     removed 1to2 bridge for topic '/chatter'
```

WebShell #2 Output: Talker ROS1

```
[ ]: [INFO] [1544551384.459151]: hello world 1544551384.46
     [INFO] [1544551384.559150]: hello world 1544551384.56
     [INFO] [1544551384.659144]: hello world 1544551384.66
```

WebShell #3 Output: Listener ROS2

```
[ ]: [INFO] [listener]: I heard: [hello world 1544551383.66]
     [INFO] [listener]: I heard: [hello world 1544551383.76]
     [INFO] [listener]: I heard: [hello world 1544551383.86]
```

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 WebShell #1: ROS1-Bridge

```
[ ]: . /home/user/.bashrc_bridge
     export ROS_MASTER_URI=http://localhost:11311
     ros2 run ros1_bridge dynamic_bridge
```

Execute in WebShell #2: Listener ROS1

```
[ ]: . /home/user/.bashrc_ros1
     rosrun roscpp_tutorials listener
```

Execute in WebShell #3: Talker ROS2

```
[ ]: . /home/user/.bashrc_ros2
     ros2 run demo_nodes_py talker
```

WebShell #1 Output: ROS1-Bridge

```
[ ]: created 2to1 bridge for topic '/chatter' with ROS 2 type
     'std_msgs/String' and ROS 1 type ''
     removed 2to1 bridge for topic '/chatter'
```

WebShell #2 Output: Listener ROS1

```
[ ]: [ INFO] [1544551475.461572300]: I heard: [Hello World: 0]
     [ INFO] [1544551476.453461509]: I heard: [Hello World: 1]
     [ INFO] [1544551477.453674267]: I heard: [Hello World: 2]
```

WebShell #3 Output: Talker ROS2

```
[ ]: [INFO] [talker]: Publishing: "Hello World: 0"
     [INFO] [talker]: Publishing: "Hello World: 1"
     [INFO] [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 WebShell #1: ROS1-Bridge

```
[ ]: # .bashrc_bridge sources everything and was used for the compilation
     . /home/user/.bashrc_bridge
     export ROS_MASTER_URI=http://localhost:11311
     ros2 run ros1_bridge dynamic_bridge
```

Execute in WebShell #2: add_two_ints_server ROS1

```
[ ]: # .bashrc_bridge sources everything and was used for the compilation
     . /home/user/.bashrc_ros1
     export ROS_MASTER_URI=http://localhost:11311
     rosrun roscpp_tutorials add_two_ints_server
```

Execute in WebShell #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
```

WebShell #1 Output: ROS1-Bridge

```
[ ]: Created 2 to 1 bridge for service /add_two_ints
     Removed 2 to 1 bridge for service /add_two_ints
```

WebShell #2 Output: add_two_ints_server ROS1

```
[ ]: [ INFO] [1544551634.781450042]: request: x=2, y=3
     [ INFO] [1544551634.782151118]:   sending back response: [5]
```

WebShell #3 Output: add_two_ints_client ROS2

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

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 4.1

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 WebShell #1: ROS1-Bridge

```
[ ]: . /home/user/.bashrc_bridge
     export ROS_MASTER_URI=http://localhost:11311
     ros2 run ros1_bridge dynamic_bridge
```

Execute in WebShell #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 WebShell #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
```

WebShell #1 Output: ROS1-Bridge

```
[ ]: created 1to2 bridge for topic '/image' with ROS 1 type
     'sensor_msgs/Image' and ROS 2 type 'sensor_msgs/Image'
     [INFO] [ros1_bridge]: Passing message from ROS 1 sensor_msgs/Image to
     ROS 2 sensor_msgs/Image (showing msg only once per type)
```

WebShell #2 Output: Image topic remapper ROS1

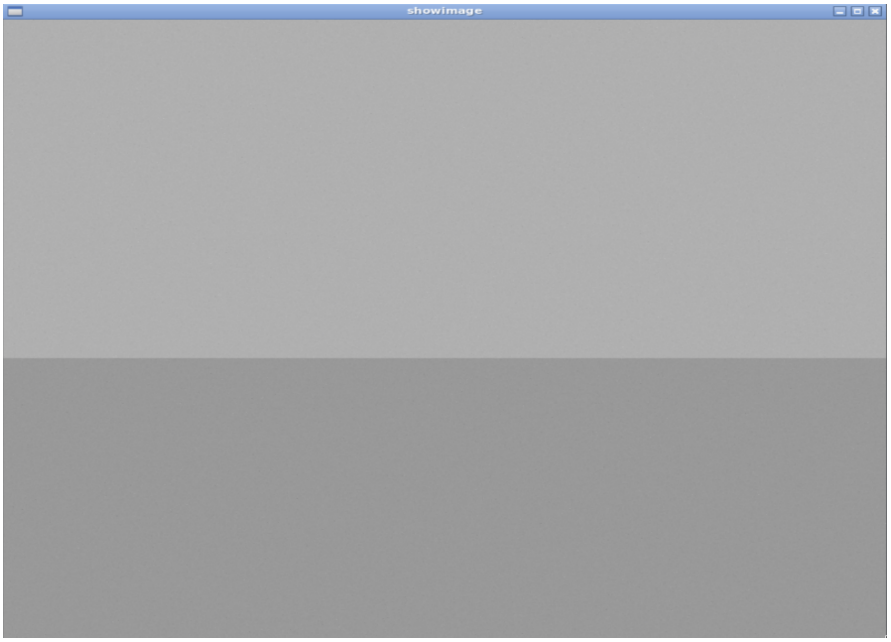
```
[ ]:
```

WebShell #3 Output: Cam2image

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

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

Go to the graphical interface window (hit the icon with a screen in the IDE)



BB8 Camera

Example 4.2

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 WebShell #1: ROS1-Bridge

```
[ ]: . /home/user/.bashrc_bridge
     export ROS_MASTER_URI=http://localhost:11311
     ros2 run ros1_bridge dynamic_bridge
```

Execute in WebShell #3: 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}}"
```

WebShell #1 Output: ROS1-Bridge

```
[ ]: created 2to1 bridge for topic '/cmd_vel' with ROS 2 type
     'geometry_msgs/Twist' and ROS 1 type ''
     [INFO] [ros1_bridge]: Passing message from ROS 2 geometry_msgs/Twist
     to ROS 1 geometry_msgs/Twist (showing msg only once per type)
```

WebShell #3 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:

BB8 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}}"
```

Example 4.3

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 WebShell #1: ROS1-Bridge

```
[ ]: . /home/user/.bashrc_bridge
     export ROS_MASTER_URI=http://localhost:11311
     ros2 run ros1_bridge dynamic_bridge
```

Execute in WebShell #2: Publish movement command through ROS2
To Make the robot turn:

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

WebShell #1 Output: ROS1-Bridge

```
[ ]: Created 2 to 1 bridge for service /bb8/camera1/set_camera_info
     Created 2 to 1 bridge for service /bb8/imu/service
     Created 2 to 1 bridge for service /gazebo/pause_physics
     Created 2 to 1 bridge for service /gazebo/reset_simulation
     Created 2 to 1 bridge for service /gazebo/reset_world
     Created 2 to 1 bridge for service /gazebo/unpause_physics
```

WebShell #2 Output: Image topic remapper ROS1

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

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

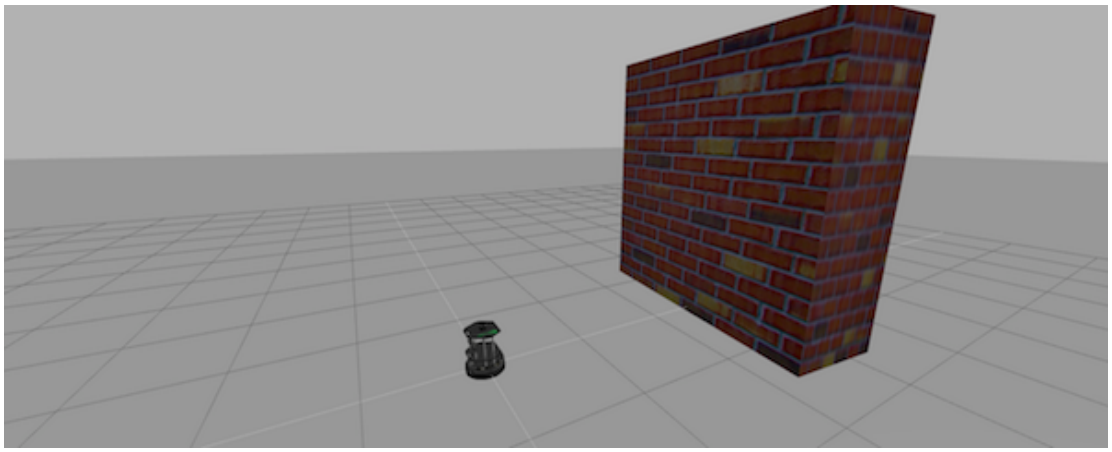
The robot should have been reset to its original state before you started tinkering with movement commands.

Congratulations, you can now combine ROS1 and ROS2.

Unit 3. Topics Part1

ROS2 BASICS IN 5 DAYS

Unit 3: Topics



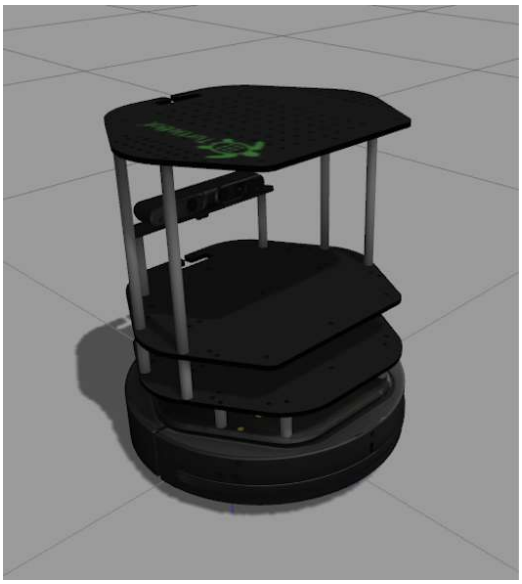
Kobuki

Run on ROSDS

- ROSject Link: https://bit.ly/2MBJDb2
- Robot: **Turtlebot 2**

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



Kobuki Robot

Part 1: Publisher

Exercise 2.1

- 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.

Data for Excercice 2.1
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/crystal/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 {2.1}: 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");
       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;
     }
```

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 WebShell #1

```
[ ]: ros2 topic list | grep  '/counter'
```

WebShell #1 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 WebShell #1

```
[ ]: ros2 topic info /counter
```

WebShell #1 Output

```
[ ]: user:~$ ros2 topic info /counter
     Topic: /counter
     Publisher count: 1
     Subscriber 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 WebShell #1

```
[ ]: ros2 topic echo /counter
```

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

WebShell #1 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
       auto publisher =
     node->create_publisher<std_msgs::msg::Int32>("counter");
       // 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
```

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 can read more about it here: https://index.ros.org/doc/ros2/Composition/
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 {2.1b}: 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");
         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;
}
```

simple_topic_publisher_composable.cpp

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");
     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!

Messages

As you may have noticed, topics handle information through messages. There are many different types of messages.

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

Messages are defined in **.msg** files, which are located inside a **msg** directory of a package.

To get information about a message, use the following command:

```
[ ]: ros2 msg show <message>
```

Example 2.1
For example, let's try to get information about the std_msgs/Int32 message. Type the following command and check the output.

Execute in WebShell #1

```
[ ]: ros2 msg show std_msgs/Int32
```

WebShell #1 Output

```
[ ]: user:~$ ros2 msg show std_msgs/Int32
     int32 data
```

In this case, the **Int32** message has only one variable of type **int32**, named **data**. This Int32 message comes from the package std_msgs, and you can find it in its **msg** directory.

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

Exercise 2.2
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.

Data for Excercice 2.2
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:

```
[ ]: 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.

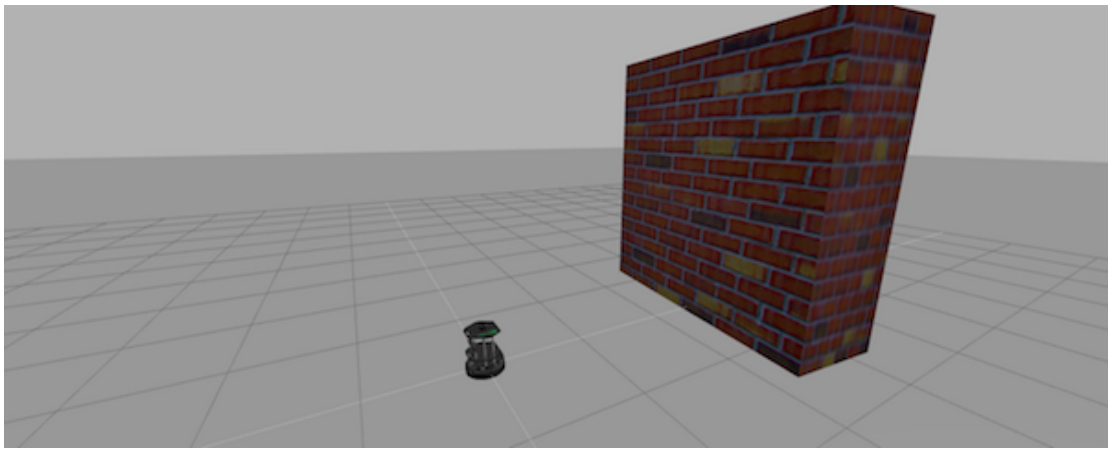
##
Solutions
Please try to do it by yourself unless you get stuck or need some inspiration. You will learn much more if you fight through each exercise.

Follow this link to open the solutions for the Topics Part 1:Topics Part 1 Solutions

Unit 3. Topics Part 2

ROS2 BASICS IN 5 DAYS

Unit 3: Topics

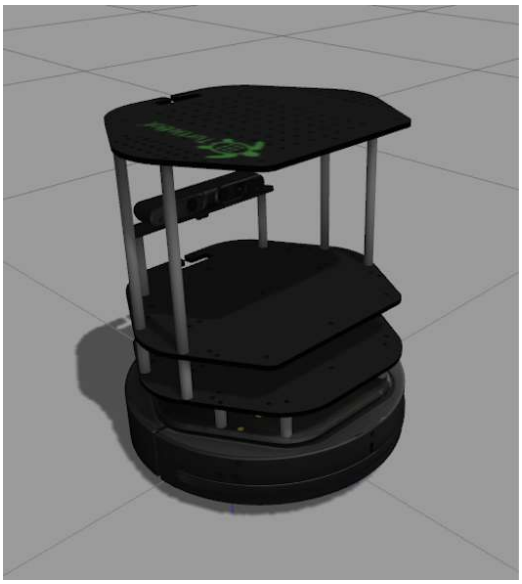


Kobuki

- ROSject Link: https://bit.ly/2MBJDb2
- Robot: **Turtlebot 2**

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 message



Kobuki Robot

Part 2: Subscriber

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:

Example 2.3

- 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.

C++ Program {2.2}: simple_topic_subscriber.cpp

```
[ ]: #include "rclcpp/rclcpp.hpp"
     #include "std_msgs/msg/int32.hpp"

     rclcpp::Node::SharedPtr g_node = nullptr;

     void topic_callback(const std_msgs::msg::Int32::SharedPtr msg)
     {
       RCLCPP_INFO(g_node->get_logger(), "I heard: '%d'", msg->data);
```

```
}

int main(int argc, char * argv[])
{
  rclcpp::init(argc, argv);
  g_node = rclcpp::Node::make_shared("simple_subscriber");
  auto subscription =
g_node->create_subscription<std_msgs::msg::Int32>
      ("counter", topic_callback);
  rclcpp::spin(g_node);
  rclcpp::shutdown();

  subscription = nullptr;
  g_node = nullptr;
  return 0;
}
```

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 WebShell #1

```
[ ]: 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 WebShell #2

```
[ ]: 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:

WebShell #1 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"

     rclcpp::Node::SharedPtr g_node = nullptr;

     // 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(g_node->get_logger(), "I heard: '%d'", msg->data);
     }

     int main(int argc, char * argv[])
     {
```

```
  rclcpp::init(argc, argv);
  // Initiate a Node called 'simple_subscriber'
  g_node = rclcpp::Node::make_shared("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
  auto subscription =
g_node->create_subscription<std_msgs::msg::Int32>
      ("counter", topic_callback);
  // Create a loop that will keep the program in execution
  rclcpp::spin(g_node);
  rclcpp::shutdown();

  subscription = nullptr;
  g_node = nullptr;
  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?

Node Composition

As explained in the previous chapter, the C++ script you have just checked, **simple_topic_subscriber.cpp**, is written using the **old school** programming method. So, it won't be able to use Node Composition. In order to be able to use Node Composition, you would have to use a script like the one below.

C++ Program {2.2b}: simple_topic_subscriber_composable.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", 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;
}
```

simple_topic_subscriber_composable.cpp
As you can see, it's quite similar to the first script, but in this case, we are using a class named **SimpleSubscriber**, which inherits from the superclass **Node** - just like in the example you saw in the previous chapter.

Now, let's do some exercises to put into practice what you've learned!

Exercise 2.4
Modify the previous code in order to print the odometry of the robot.

Data for Exercice 2.4
The odometry of the robot is published by the robot into the /odom topic.

You will need to figure out what message uses the /odom topic, and the structure of this message.

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.

Solution Exercise 2.4
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
Exercise 2.5

1. Add to {Exercice 2.4} 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 message, called Age.msg. See the detailed description How to prepare CMakeLists.txt and package.xml for custom topic message compilation.

Solution Exercise 2.5
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

How to Prepare CMakeLists.txt and package.xml for Custom Topic Message Compilation

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 ROS defined (**ros2 msg 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 a message 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 four 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(builtin_interfaces 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 file should look like this.

```
[ ]: rosidl_generate_interfaces(new_msg
       "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 topic_ex, 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(builtin_interfaces 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(new_msg
  "msg/Age.msg"
)

ament_package()
```

4) Modify package.xml

First, you will need to set the **package format** to 3. Note that, by default, this will be set to 2, so you will need to manually modify it.

```
[ ]: <package format="3">
```

This has to be done because the **member_of_group command** requires format 3.

Now, just add the following lines to the package.xml file.

```
[ ]: <build_depend>builtin_interfaces</build_depend>
     <build_depend>rosidl_default_generators</build_depend>
     <exec_depend>builtin_interfaces</exec_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>builtin_interfaces</build_depend>
       <build_depend>rosidl_default_generators</build_depend>
       <exec_depend>builtin_interfaces</exec_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 WebShell #1

```
[ ]: cd ~/ros2_ws
     colcon build --symlink-install --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 msg show new_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 WebShell #1

```
[ ]: ros2 msg show new_msg/Age
```

WebShell #1 Output

```
[ ]: user ~ $ ros2 msg show new_msg/Age
     float32 years
     float32 months
     float32 days
```

To use Custom Messages in Cpp files

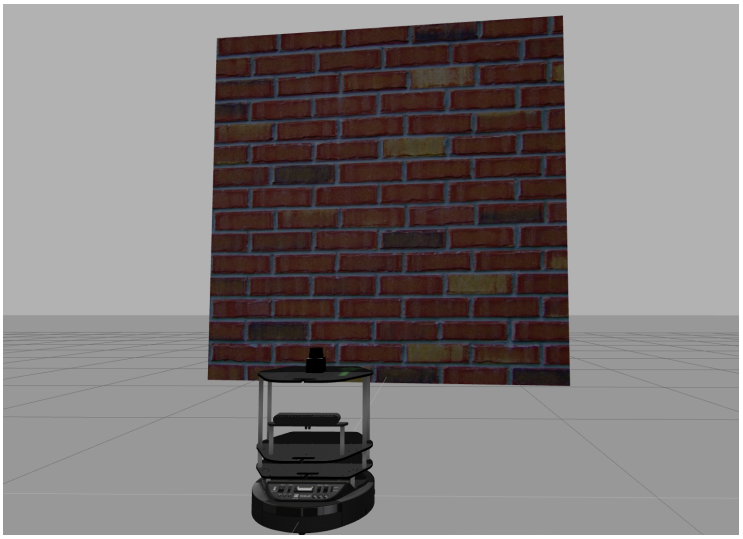
You will have to add to your **CMakeLists.txt** the following extra lines to compile and link your executable (in this example, it's called **publish_age.cpp**):

```
[ ]: find_package(new_msg REQUIRED)

     add_executable(age_publisher_node src/publish_age.cpp)
     ament_target_dependencies(age_publisher_node rclcpp std_msgs new_msg)

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

Topics Mini Project



Tultlebot2

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 small project, you will create a code to make the robot avoid the wall that is in front of it. To help you achieve this, let's divide the project 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.

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 WebShell #1

```
[ ]: ros2 msg show sensor_msgs/LaserScan
```

WebShell #1 Output

```
[ ]: user ~ $ ros2 msg show sensor_msgs/LaserScan
     std_msgs/Header header
       uint32 seq
       time stamp
       string frame_id
     float32 angle_min
     float32 angle_max
     float32 angle_increment
     float32 time_increment
     float32 scan_time
     float32 range_min
     float32 range_max
     float32[] ranges <-- Use only this one
     float32[] intensities
```

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).

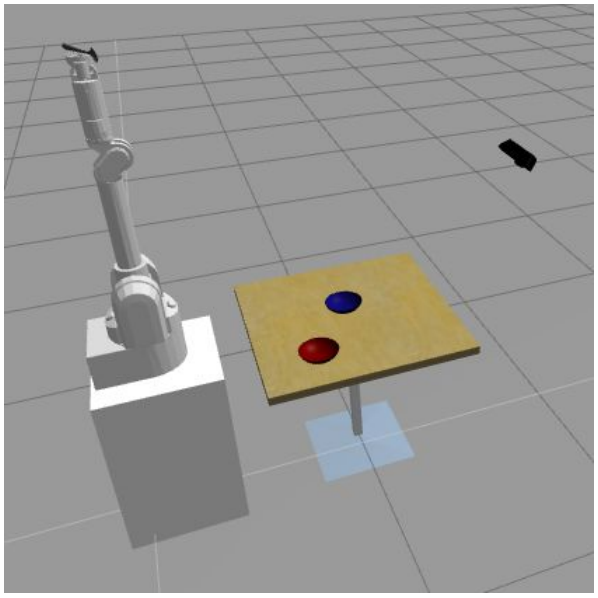
HINT 4: The scope of the laser is about 180 degrees from right to left. This means that the values at the beginning and at the end of the 'ranges' array will be the ones related to the readings on the sides of the laser (left and right), while the values in the middle of the array will be

the ones related to the front of the laser.

Unit 4. Services in ROS Part 1

ROS2 BASICS IN 5 DAYS

Cpp Services in ROS2 Part 1



Iri Wam Robot

- ROSject Link: https://bit.ly/2FPf3tR
- Robot: **WAM Arm**

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

Part 1

Congratulations! You now know 75% of ROS Basics! With topics, you can do more or less whatever you want and need for your astromech droid. Many ROS 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.

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 ROS systems, such as the navigation system. The face-recognition system will provide a Service. Your ROS program will call that service and WAIT until it gives you 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 teh robot moves toward the coordinates.

So... what's the difference between a Service and an Action? Services are Synchronous. When your ROS program calls a service, your program can't continue until it receives a result from the service. Actions are Asynchronous. It's like launching a new thread. When your ROS program calls an action, your program can perform other tasks while the action is being performed in another thread.

Conclusion: Use services when your program can't continue until it receives the result from the service.

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.

Get the structure of service messages and commands:

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 srv list**: Lists all the Service messages available
- **ros2 srv package**: Lists all the service messages from a package
- **ros2 srv packages**: Lists all the available packages that **have** service messages defined inside them.
- **ros2 srv 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.

ROS2 service list

Execute in WebShell #1
Remember that whenever you interact with ROS1, you need to launch **ROS1-Bridge**.

```
[ ]: . /home/user/.bashrc_bridge
     export ROS_MASTER_URI=http://localhost:11311
     ros2 run ros1_bridge dynamic_bridge
```

Execute in WebShell #2
With this command, you can get all the services running now. In this case, most of them are related to ROS1-Gazebo.

```
[ ]: # Get all the services currently running
     . /home/user/.bashrc_ros2
     ros2 service list
```

WebShell #2 Output

```
[ ]: /camera/set_camera_info
     /gazebo/pause_physics
```

```
/gazebo/reset_simulation
/gazebo/reset_world
/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
```

Limitations of ROS1-Bridge and how to circumvent them

Here comes a very important topic, which is the Support that **ROS1-Bridge** debians have towards **service messages**. For the moment, not all messages are supported. This means that although both types of messages are installed both in the **melodic ROS1** and **crystal ROS2**, because the ROS1-Bridge wasn't compiled versus those messages, it won't be able to connect them!

A good way to see if ROS1-Bridge supports the service is the list of services given at the start. In this case, you can see that only these are supported:

```
[ ]: /camera/set_camera_info
     /gazebo/pause_physics
     /gazebo/reset_simulation
     /gazebo/reset_world
     /gazebo/unpause_physics
```

This is because these are the only services in Gazebo for ROS1 that don't use service messages from the **gazebo_msgs** type.

For example, let's check the type of message used in the service /**gazebo**/**delete_model** in **ROS1**
Execute in WebShell #3

```
[ ]: . /home/user/.bashrc_ros1
     rosservice info /gazebo/delete_model
```

Output of WebShell #3

```
[ ]: Node: /gazebo
     URI: rosrpc://10.8.0.1:35755
     Type: gazebo_msgs/DeleteModel
     Args: model_name
```

This is why the ROS1-Bridge doesn't find it and you won't be able to send anything to that service from ROS2 to ROS1.

So, in this scenario, you have three options:

- You compile ROS1-Bridge from source with the new messages in the same workspace. We won't to this in this basic course.
- You use an entire ROS2 system, not needing ROS1-Bridge. We will do that in the **MARA** robot.
- You create **Bridge Services** that are created and launched in ROS1 and use supported ROS1-Bridge messages; in this case **std_srvs**. This service will be able to communicate with ROS2 through ROS1-Bridge and then execute the equivalent action in ROS1 space. This is what we are going to do in this course.

Let's see an example:

ROS2 service call

We want to call a **ROS1** service of Gazebo to delete an object in the scene. Because it's **ROS1 Gazebo**, we need **ROS1-Bridge** to interact with it.

```
[ ]: ros2 service call /gazebo/delete_model gazebo_msgs/DeleteModel
     '{model_name: TestingName}'
```

Execute in WebShell #1
Remember that whenever you interact with ROS1, you need to launch **ROS1-Bridge**.

```
[ ]: . /home/user/.bashrc_bridge
     export ROS_MASTER_URI=http://localhost:11311
     ros2 run ros1_bridge dynamic_bridge
```

Execute in WebShell #2

```
[ ]: # Call A Service Server
     . /home/user/.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 YAMLCommandLine 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.

WebShell #2 Output

```
[ ]: waiting for service to become available...
```

As you can see, nothing happens. That's because that topic is NOT in the ROS2 space, due to the limitations mentioned about **ROS1-Bridge**. Therefore, you will have to create a **Bridge-Service**

in ROS1.

If you don't control **ROS1 Basics**, we recommend following the **ROS in Five Days with CPP** course before continuing to the next step.

Execute in WebShell #3 ROS1

```
[ ]: . /home/user/.bashrc_ros1
     cd ~/catkin_ws/src
     catkin_create_pkg my_bridge_ros1_pkgs roscpp std_srvs
     touch my_bridge_ros1_pkgs/src/bridge_delete_model_server.cpp
     cd ~/catkin_ws
     catkin_make
     source devel/setup.bash
     rospack profile
```

We create the following files and compile:

C++ Program {3.1}: bridge_delete_model_server.cpp

```
[ ]: #include <ros/ros.h>
     #include "gazebo_msgs/DeleteModel.h"
     #include <std_srvs/Empty.h>

     class BridgeDeleteModelServer
     {
         private:
             // ROS Objects
             ros::NodeHandle nh_;

             // ROS Services
             ros::ServiceServer srv_perform_square_;

             ros::ServiceServer bridge_delete_model_service_server_;
             ros::ServiceClient delete_model_service_client_;



         public:

             BridgeDeleteModelServer()
             {
                 ROS_INFO("Creating Service...");
                  // create the Service called
                 this->bridge_delete_model_service_server_ =
     nh_.advertiseService("/my_bridge_delete_model",
                          &BridgeDeleteModelServer::my_callback,
                          this);
```

```
            this->delete_model_service_client_ =
nh_.serviceClient<gazebo_msgs::DeleteModel>("/gazebo/delete_model");
            ROS_INFO("Creating Service...DONE");

        }

        ~BridgeDeleteModelServer(void)
        {

        }

        bool my_callback(std_srvs::Empty::Request &req,
                         std_srvs::Empty::Response &res)
        {
            // Create an object of type DeleteModel
            gazebo_msgs::DeleteModel srv;
            // Fill the variable model_name of this object with the
desired value
            srv.request.model_name = "bowl_1";

            // Send through the connection the name of the object to
be deleted by the service
            if (delete_model_service_client_.call(srv))
            {
                // Print the result given by the service called
                ROS_INFO("%s", srv.response.status_message.c_str());
                return true;
            }
            else
            {
                ROS_ERROR("Failed to call service delete_model");
                return false;
            }


        }

};

int main(int argc, char** argv)
{
  ros::init(argc, argv, "bridge_delete_model_server_node");

  BridgeDeleteModelServer bridge_delete;

  ros::spin();
```

```
  return 0;
}
```

The only line that you should be concerned with here is the one where we specify that the model be deleted:

```
[ ]: srv.request.model_name = "bowl_1";
```

And change the **CMakelists.txt** and **package.xml**
CMake File {3.1}: CMakelists.txt

```
[ ]: cmake_minimum_required(VERSION 2.8.3)
     project(my_bridge_ros1_pkgs)

     ## Compile as C++11, supported in ROS Kinetic and newer
     # add_compile_options(-std=c++11)

     ## Find catkin macros and libraries
     ## if COMPONENTS list like find_package(catkin REQUIRED COMPONENTS
     xyz)
     ## is used, also find other catkin packages
     find_package(catkin REQUIRED COMPONENTS
       roscpp
       std_srvs
       gazebo_msgs
     )


     catkin_package(
     #  INCLUDE_DIRS include
     #  LIBRARIES my_bridge_ros1_pkgs
     )


     include_directories(
     # include
       ${catkin_INCLUDE_DIRS}
     )

     add_executable(bridge_delete_model_server_node
     src/bridge_delete_model_server.cpp)
     add_dependencies(bridge_delete_model_server_node
     ${bridge_delete_model_server_node_EXPORTED_TARGETS}
     ${catkin_EXPORTED_TARGETS})
     target_link_libraries(bridge_delete_model_server_node
        ${catkin_LIBRARIES}
      )
```

XML File {3.1}: package.xml

```
[ ]: <?xml version="1.0"?>
     <package format="2">
       <name>my_bridge_ros1_pkgs</name>
       <version>0.0.0</version>
       <description>The my_bridge_ros1_pkgs package</description>
       <maintainer email="user@todo.todo">user</maintainer>
       <license>TODO</license>

       <buildtool_depend>catkin</buildtool_depend>
       <build_depend>roscpp</build_depend>
       <build_depend>std_srvs</build_depend>
       <build_depend>gazebo_msgs</build_depend>
       <build_export_depend>roscpp</build_export_depend>
       <build_export_depend>std_srvs</build_export_depend>
       <build_export_depend>gazebo_msgs</build_export_depend>
       <exec_depend>roscpp</exec_depend>
       <exec_depend>std_srvs</exec_depend>
       <exec_depend>gazebo_msgs</exec_depend>



       <!-- The export tag contains other, unspecified, tags -->
       <export>
         <!-- Other tools can request additional information be placed here
     -->

       </export>
     </package>
```

Execute in WebShell #3 ROS1
We compile the new program:

```
[ ]: . /home/user/.bashrc_ros1
     cd ~/catkin_ws
     catkin_make
     source devel/setup.bash
     rospack profile
```

Ok, now that we have all the pieces to make this work, let's return and launch everything in the corresponding webshells:

Execute in WebShell #1
Remember that whenever you interact with ROS1, you need to launch **ROS1-Bridge**.

```
[ ]: . /home/user/.bashrc_bridge
     export ROS_MASTER_URI=http://localhost:11311
     ros2 run ros1_bridge dynamic_bridge
```

Execute in WebShell #3 ROS1 Service that we created in ROS1

```
[ ]: . /home/user/.bashrc_ros1
     rosrun my_bridge_ros1_pkgs bridge_delete_model_server_node
```

Execute in WebShell #2

```
[ ]: # We check that the bridge service is now available for ROS2 space
     . /home/user/.bashrc_ros2
     ros2 service list | grep /my_bridge_delete_model
```

WebShell #2 Output: ROS2

```
[ ]: /my_bridge_delete_model
```

If you got the name of the topic in the output, /**my_bridge_delete_model**, then everything went ok and you can now proceed to call that service and **Remove the bowl_1 model** from the scene with ROS2.

Execute in WebShell #2 ROS2

```
[ ]: # Call A Service Server
     . /home/user/.bashrc_ros2
     ros2 service call /my_bridge_delete_model std_srvs/Empty '{}'
```

WebShell #2 Output: ROS2

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

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

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.

Bowls

ROS2 srv list

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

Execute in WebShell #2
Note that in the following commands, bridge is not necessary because it's only for message information and the ROS2 system, in general.

```
[ ]: # List all available service messages in the system
     . /home/user/.bashrc_ros2
     ros2 srv list
```

WebShell #2 Output

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

As you can see in the available message types, you **CAN use Gazebo_msgs**, but that doesn't mean that the debian from ROS2 bridge can use them because it would have to be compiled versus those messages, which seems not to be the case.

ROS2 srv package gazebo_msgs

Execute in WebShell #2

```
[ ]: # List all the messages defined in a certain package
     . /home/user/.bashrc_ros2
     ros2 srv package gazebo_msgs
```

WebShell #2 Output

```
[ ]: gazebo_msgs/ApplyBodyWrench
     gazebo_msgs/ApplyJointEffort
     gazebo_msgs/BodyRequest
     gazebo_msgs/DeleteEntity
```

```
gazebo_msgs/DeleteLight
gazebo_msgs/DeleteModel
gazebo_msgs/GetJointProperties
gazebo_msgs/GetLightProperties
gazebo_msgs/GetLinkProperties
gazebo_msgs/GetLinkState
gazebo_msgs/GetModelProperties
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
gazebo_msgs/SetModelConfiguration
gazebo_msgs/SetModelState
gazebo_msgs/SetPhysicsProperties
gazebo_msgs/SpawnEntity
gazebo_msgs/SpawnModel
```

Execute in WebShell #2

```
[ ]: # List all the packages that have Service Messages defined in them
     . /home/user/.bashrc_ros2
     ros2 srv packages
```

WebShell #2 Output

```
[ ]: composition
     diagnostic_msgs
     example_interfaces
     gazebo_msgs
     lifecycle_msgs
     logging_demo
     map_msgs
     nav_msgs
     rcl_interfaces
     sensor_msgs
     std_srvs
     tf2_msgs
     unit_3_services_custom_msgs
```

Execute in WebShell #2

```
[ ]: # List all the messages defined in a certain package
     . /home/user/.bashrc_ros2
     ros2 srv show gazebo_msgs/DeleteModel
```

WebShell #2 Output

```
[ ]: 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
```

And here we can introduce the structure of the **Service Messages**. Does this **Output for the DeleteModel message** seem familiar? It should because it's the same structure as the Topics messages, 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...
Example 3.1

Create cpp_unit_3_services package

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

Execute in WebShell #1

```
[ ]: . /home/user/.bashrc_ros2
     cd ~/ros2_ws/src
     ros2 pkg create cpp_unit_3_services --build-type ament_cmake
     --dependencies std_msgs rclcpp gazebo_msgs
```

WebShell #1 Output

```
[ ]: going to create a new package
     package name: cpp_unit_3_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_3_services
     creating ./cpp_unit_3_services/package.xml
     creating source and include folder
     creating folder ./cpp_unit_3_services/src
     creating folder ./cpp_unit_3_services/include/cpp_unit_3_services
     creating ./cpp_unit_3_services/CMakeLists.txt
```

And now we compile.

Execute in WebShell #1

```
[ ]: . /home/user/.bashrc_ros2
     cd ~/ros2_ws
     # Compile all workspace
     # colcon build --symlink-install
     # Compile only the package we have created
     colcon build --symlink-install --packages-select cpp_unit_3_services
```

WebShell #1 Output

```
[ ]: Starting >>> cpp_unit_3_services
     Finished <<< cpp_unit_3_services [2.54s]

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

Create Service Client and Dummy Server

Let's now learn how to call and give a service in ROS2. Let's start with a **dummy Server**.

Execute in WebShell #1

```
[ ]: cd ~/ros2_ws/src/cpp_unit_3_services
     touch src/cpp_simple_service_client.cpp
```

C++ Program {3.2}: 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_1";

    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;
}
```

Execute in WebShell #1

```
[ ]: cd ~/ros2_ws/src/cpp_unit_3_services
     touch src/cpp_simple_service_dummy_server.cpp
```

C++ Program {3.4}: cpp_simple_service_dummy_server.cpp

```
[ ]: #include <inttypes.h>
     #include <memory>
     #include "rclcpp/rclcpp.hpp"
     #include "gazebo_msgs/srv/delete_model.hpp"
```

```
using DeleteModel = gazebo_msgs::srv::DeleteModel;
rclcpp::Node::SharedPtr g_node = nullptr;

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)
{
    (void)request_header;
    RCLCPP_INFO(
    g_node->get_logger(),
    "Incoming request\nModel-To_Delete-Name: %s",
request->model_name.c_str());

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

int main(int argc, char ** argv)
{
    rclcpp::init(argc, argv);
    g_node = rclcpp::Node::make_shared("cpp_simple_service_server");
    auto server =
g_node->create_service<DeleteModel>("/gazebo/delete_model",
handle_service);
    rclcpp::spin(g_node);
    rclcpp::shutdown();
    g_node = nullptr;
    return 0;
}
```

cpp_simple_service_dummy_server.cpp

Now, we make the necessary changes to the **CMakeLists.txt** to compile the **client** and **dummy_server** and install them in our workspace.

Setup {3.4}: CMakeLists.txt

```
[ ]: cmake_minimum_required(VERSION 3.5)
project(cpp_unit_3_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()

function(custom_executable target)
  add_executable(${target}_node src/${target}.cpp)
  ament_target_dependencies(${target}_node
    "gazebo_msgs"
    "rclcpp"
    "std_msgs")
  install(TARGETS ${target}_node
  DESTINATION lib/${PROJECT_NAME})
endfunction()

# Adding Services
custom_executable(cpp_simple_service_client)
custom_executable(cpp_simple_service_dummy_server)

ament_package()
```

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

Execute in WebShell #2

```
[ ]: . /home/user/.bashrc_ros2
     cd ~/ros2_ws
```

```
# We compile everythin in the ws except the packages we dont want.
colcon build --symlink-install --packages-skip
NAME_OF_NONCOMPILE_PACKAGE

# Or compile only the package when you do changes:
colcon build --symlink-install --packages-select cpp_unit_3_services
```

Execute in WebShell #2

```
[ ]: # Check that your Client And server binaries were generated
     ll ~/ros2_ws/install/cpp_unit_3_services/lib/cpp_unit_3_services/cpp_s
     imple_service_client_node
     ll ~/ros2_ws/install/cpp_unit_3_services/lib/cpp_unit_3_services/cpp_s
     imple_service_dummy_server_node
```

WebShell #1 Output

```
[ ]: lrwxrwxrwx 1 user user 75 Dec 19 18:32 /home/user/ros2_ws/install/cpp_
     unit_3_services/lib/cpp_unit_3_services/cpp_simple_service_client_node
     -> /home/user/ros2_ws/build/cpp_unit_3_services/cpp_simple_service_cli
     ent_node*
     lrwxrwxrwx 1 user user 81 Dec 19 18:32 /home/user/ros2_ws/install/cpp_
     unit_3_services/lib/cpp_unit_3_services/cpp_simple_service_dummy_serve
     r_node -> /home/user/ros2_ws/build/cpp_unit_3_services/cpp_simple_serv
     ice_dummy_server_node*
```

As you can see, the executables that will be run are softlinks to the binaries build, found in the **build** folder. Remember that we will always execute the install elements, not the build directly.

And now, we compile:

Now, let's execute both **client/dummy-server** to check that they work:

Execute in WebShell #2: ROS2 Service CLIENT
ROS2 doesn't always have the **DoubleTab** completion working at the moment, but in this case, you **CAN** use it for auto completing and checking that the system correctly finds the path to the executable.

```
[ ]: . /home/user/.bashrc_ros2
     ros2 run cpp_unit_3_services cpp_simple_service_client_node
```

Execute in WebShell #3: ROS2 Service Server Dummy

```
[ ]: . /home/user/.bashrc_ros2
     ros2 run cpp_unit_3_services cpp_simple_service_dummy_server_node
```

WebShell #1 Output

```
[ ]: [INFO] [cpp_simple_service_client]: service not available, waiting
     again...
     [INFO] [cpp_simple_service_client]: service not available, waiting
     again...
     [[INFO] [cpp_simple_service_client]: Client request->model_name :
     bowl_1
     [INFO] [cpp_simple_service_client]: Result-Success : True
     [INFO] [cpp_simple_service_client]: Result-Status: The Model bowl_1
     was deleted.
```

The **service not available** will be prompted during the time that you don't launch the server
WebShell #2 Output

```
[ ]: [INFO] [cpp_simple_service_server]: Incoming request
     Model-To_Delete-Name: bowl_1
```

Exercise 3.1
Great, now it's time to test it in the real simulation. For that, we will have to reuse the program that we did to be able to communicate through standard services for the reasons mentioned then. So, you will have to do two things:

- Change the name of the service to connect to in the cpp_simple_service_client_node
- Change the type of service messages used, because remember that the service uses std_srvs/Empty.srv messages.
- Change the model to be deleted in the **bridge_delete_model_server.cpp** and recompile in ROS1.

```
[ ]: service_name = '/dummy_gazebo/delete_model' --> service_name =
     '/my_bridge_delete_model'
     service_message_type = 'gazebo_msgs/DeleteModel.srv' -->
     service_message_type = 'std_srvs/Empty.srv'
     Model to remove now --> "cafe_table"
```

Execute in WebShell #1
Remember that whenever you interact with ROS1, you need to launch **ROS1-Bridge**.

```
[ ]: . /home/user/.bashrc_bridge
     export ROS_MASTER_URI=http://localhost:11311
     ros2 run ros1_bridge dynamic_bridge
```

Execute in WebShell #2 ROS1 Service that we created in ROS1

```
[ ]: . /home/user/.bashrc_ros1
     rosrun my_bridge_ros1_pkgs bridge_delete_model_server_node
```

Execute in WebShell #3

```
[ ]: . /home/user/.bashrc_ros2
     ros2 run cpp_unit_3_services cpp_simple_service_client_ex3_1_node
```

##
Solutions
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:Cpp Services Part 1 Solutions
Now, let's create a launch file to do the exact same thing you did in **Exercise 3.1**:

Execute in WebShell #1

```
[ ]: cd ~/ros2_ws/src/cpp_unit_3_services
     mkdir launch
     touch launch/start_cpp_simple_service_client_ex3_1.cpp.launch.py
     chmod +x launch/start_cpp_simple_service_client_ex3_1.cpp.launch.py
```

C++ Program {3.5}: start_ cpp_simple_service_client_ex3_1.cpp.launch.py

```
[ ]: """Launch cpp_simple_service_client_node_ex3_1"""

     from launch import LaunchDescription
     import launch_ros.actions


     def generate_launch_description():
         return LaunchDescription([
             launch_ros.actions.Node(
                 package='cpp_unit_3_services',
     node_executable='cpp_simple_service_client_ex3_1_node',
     output='screen'),
         ])
```

cpp_simple_service_client_ex3_1.cpp.launch.py
And add the following line to to the **CMakelists.txt**:

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

We now compile and execute:

Execute in WebShell #1

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

```
[ ]: . /home/user/.bashrc_bridge
     export ROS_MASTER_URI=http://localhost:11311
     ros2 run ros1_bridge dynamic_bridge
```

Execute in WebShell #2 ROS1 Service that we created in ROS1

```
[ ]: . /home/user/.bashrc_ros1
     rosrun my_bridge_ros1_pkgs bridge_delete_model_server_node
```

Execute in WebShell #3

```
[ ]: . /home/user/.bashrc_ros2
     cd ~/ros2_ws
     colcon build --symlink-install --packages-select cpp_unit_3_services
     ros2 launch cpp_unit_3_services
     start_cpp_simple_service_client_ex3_1.cpp.launch.py
```

WebShell #3 Output

```
[ ]: [INFO] [launch]: process[cpp_simple_service_client_ex3_1_node-1]:
     started with pid [25863]
     [INFO] [cpp_simple_service_client_ex3_1]: service not available,
     waiting again...
     [INFO] [cpp_simple_service_client_ex3_1]: service not available,
     waiting again...
     [INFO] [cpp_simple_service_client_ex3_1]: Client Requested to remove
     model.
     [INFO] [cpp_simple_service_client_ex3_1]: Result-Success.
     [INFO] [launch]: process[cpp_simple_service_client_ex3_1_node-1]:
     process has finished cleanly
```

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

Unit 4. Services in ROS Part 2

ROS2 BASICS IN 5 DAYS

Services in ROS2 Part 2

BB8 Robot

- ROSject Link: https://bit.ly/2FKJkd6
- Robot: **BB-8**

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

Part 2: How to give a Service

Here we will reinforce what you already did briefly in previous parts of this unit, which is create a server.

Example 3.7
This is what we are going to do in this example:

- Inside the src folder of the package **cpp_unit_3_services**, which was created in a previous unit, create a new file named **empty_service_server.cpp**.
- Create a service server that uses **std_srvs/Empty.srv** messages, and when called, it prints a log-info message. We will include that message by: **#include "std_srvs/srv/empty.h"**. The installed includes will be found here: **/opt/ros/bouncy/include/std_srvs**.
- 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 WebShell #1

```
[ ]: cd ~/ros2_ws/src/cpp_unit_3_services
     touch src/empty_service_server.cpp
```

C++ Program {3.2}: 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;
}
```

Now, we make the necessary changes to the **CMakeLists.txt** to compile it. For that, just add this new line in the CMakelists.txt:

```
[ ]: custom_executable(empty_service_server)
```

And add some dependecies related to **std_srvs**

```
[ ]: find_package(std_srvs REQUIRED)
     ...
     add_executable(${target}_node src/${target}.cpp)
       ament_target_dependencies(${target}_node
         "gazebo_msgs"
         "rclcpp"
         "std_msgs"
         "std_srvs")
     ...
```

Your CmakeLists.txt should look something like this:

Setup {3.1}: CMakeLists.txt

```
[ ]: cmake_minimum_required(VERSION 3.5)
     project(cpp_unit_3_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
  # 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()

function(custom_executable target)
  add_executable(${target}_node src/${target}.cpp)
  ament_target_dependencies(${target}_node
    "gazebo_msgs"
    "rclcpp"
    "std_msgs"
    "std_srvs")
  install(TARGETS ${target}_node
  DESTINATION lib/${PROJECT_NAME})
endfunction()

# Adding Services
custom_executable(cpp_simple_service_client)
custom_executable(cpp_simple_service_dummy_server)
custom_executable(cpp_simple_service_client_ex3_1)
custom_executable(empty_service_server)

ament_package()

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

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

```
[ ]: <depend>std_srvs</depend>
```

Your package.xml should look something like this:

Setup {3.2}: 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_3_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>
```

And now, we compile the whole **cpp_unit_3_services** to update the changes:

Execute in WebShell #1

```
[ ]: . /home/user/.bashrc_ros2
     cd ~/ros2_ws
     colcon build --symlink-install --packages-select cpp_unit_3_services
```

WebShell #1 Output

```
[ ]: Starting >>> cpp_unit_3_services
     --- stderr: cpp_unit_3_services
     /home/user/ros2_ws/src/cpp_unit_3_services/src/empty_service_server.cp
     p: 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_3_services/src/empty_service_server.cp
p:11:43: warning: unused parameter `request' [-Wunused-parameter]
     const std::shared_ptr<Empty::Request> request,
                                           ^~~~~~~
/home/user/ros2_ws/src/cpp_unit_3_services/src/empty_service_server.cp
p:12:44: warning: unused parameter `response' [-Wunused-parameter]
     const std::shared_ptr<Empty::Response> response)
                                            ^~~~~~~~
---
Finished <<< cpp_unit_3_services [2.20s]

Summary: 1 package finished [2.32s]
  1 package had stderr output: cpp_unit_3_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.

Check that the binary was generated from your new **server**:

Execute in WebShell #1

```
[ ]: # Check that your Client And server binaries were generated
     ll ~/ros2_ws/install/cpp_unit_3_services/lib/cpp_unit_3_services/empty
     _service_server_node
```

WebShell #1 Output

```
[ ]: lrwxrwxrwx 1 ubuntu ubuntu 72 Dec 10 18:42 /home/ubuntu/ros2_ws/instal
     l/cpp_unit_3_services/lib/cpp_unit_3_services/empty_service_server_nod
     e -> /home/ubuntu/ros2_ws/build/cpp_unit_3_services/empty_service_serv
     er_node*
```

Execute in WebShell #1

```
[ ]: . /home/user/.bashrc_ros2
     ros2 run cpp_unit_3_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 do a **rosservice list**, you will be able to visualize this service among the list of available services.

Execute in WebShell #2

```
[ ]: . /home/user/.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, call the /**my_service** service manually. Remember the calling structure discussed in the previous chapter and don't forget that in this version of ROS2, **TAB-TAB to autocomplete the structure** doesn't work 100% for the Service messages. It will find that /**my_service** uses **std_srvs**/**Empty** messages, but it won't automaticaly create the message to be sent.

Execute in WebShell #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.

WebShell #1 Output

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

WebShell #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 **dummy_server** you created in a 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.";
```

As you can see, you don't **explicitly return anything**. That's because you are using a pointer to that variable that you can write and it will be updated outside immediately.

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)
```

Exercise 3.2

- The objective of this exercise is to create a service that, when called, makes BB8 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-3.1}. This class could be called, for reference, **MoveBB8**. And the C++ file that contains it could be called **move_bb8.cpp**. To move the BB8 robot, you just have to write into the /**cmd_vel** topic, as you did in the Topics Units. Bear in mind that although this is a simulation, BB8 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.
- 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 WebShell, BB8 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.

Fig.3.1 - BB8 Square Movement Diagram

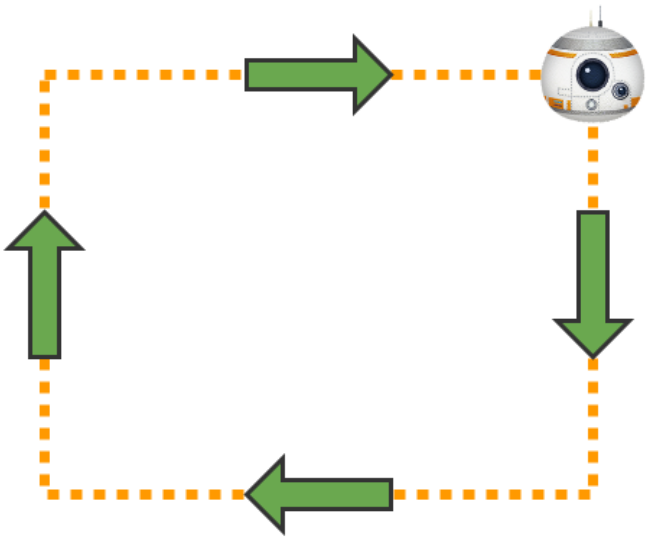


Fig.3.1 - BB8 Square Movement Diagram

Solution Exercise 3.2
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 the Services Part 2:Services Part 2 Solutions

How to create your own service message

Create a New Package

In ROS2, the custom message topics and services have to be created in a **CPP Package**. It could be done in the same package we have been using for all the examples. But we are going to create a **new package**, so we can practice importing from different packages.

So, we will create a new package for the creation of our custom service messages:

Execute in WebShell #1

```
[ ]: . /home/user/.bashrc_ros2
     cd ~/ros2_ws/src
     ros2 pkg create unit_3_services_custom_msgs --dependencies std_msgs
     rclcpp
     cd ~/ros2_ws
     # We compile only our unit_3_services_custom_msgs package, nothing
```

```
more
colcon build --symlink-install --packages-select
unit_3_services_custom_msgs
```

WebShell #1 Output

```
[ ]: going to create a new package
     package name: unit_3_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_3_services_custom_msgs
     creating ./unit_3_services_custom_msgs/package.xml
     creating source and include folder
     creating folder ./unit_3_services_custom_msgs/src
     creating folder
     ./unit_3_services_custom_msgs/include/unit_3_services_custom_msgs
     creating ./unit_3_services_custom_msgs/CMakeLists.txt
```

Create a Custom Service Message

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 WebShell #1

```
[ ]: cd ~/ros2_ws/src/unit_3_services_custom_msgs
     mkdir srv
     touch srv/MyCustomServiceMessage.srv
```

EXTRA {3.6}: 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?
```

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 messages wrappers. Also add dependencies that your custom message needs.
- package.xml: Add the dependencies that your custom message needs

Setup {3.2}: CMakeLists.txt

```
[ ]: cmake_minimum_required(VERSION 3.5)
     project(unit_3_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(builtin_interfaces 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(unit_3_services_custom_msgs
                                "srv/MyCustomServiceMessage.srv"

     ament_package()
```

We needed to add the following packages, responsible for the topics and services message generators in ROS2:

```
[ ]: find_package(builtin_interfaces REQUIRED)
     find_package(rosidl_default_generators REQUIRED)

     rosidl_generate_interfaces(unit_3_services_custom_msgs
                                "srv/MyCustomServiceMessage.srv"
```

Setup {3.2}: 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_3_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>
       <depend>builtin_interfaces</depend>
       <depend>rosidl_default_generators</depend>

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

       <member_of_group>rosidl_interface_packages</member_of_group>

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

We needed to add the following dependencies, which are for message generation and for any dependencies added in the custom message:

```
[ ]: <depend>builtin_interfaces</depend>
     <depend>rosidl_default_generators</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_generat
     e_interfaces.cmake:129 (message):
       Packages installing interfaces must include
       '<member_of_group>rosidl_interface_packages</member_of_group>' in
     their
       package.xml
```

And we have to change the **package version to 3**, otherwise we won't be able to use **member_group**.

```
[ ]: <package format="3">
```

More info: Documentation

Compile and generate the Custom Messages

Execute in WebShell #1

```
[ ]: . /home/user/.bashrc_ros2
     cd ~/ros2_ws
     # We compile only our unit_3_services_custom_msgs package, nothing
     more
     colcon build --symlink-install --packages-select
     unit_3_services_custom_msgs
```

WebShell #1 Output

```
[ ]: Starting >>> unit_3_services_custom_msgs
     [Processing: unit_3_services_custom_msgs]
     Finished <<< unit_3_services_custom_msgs [31.0s]
```

We check that the messages were generated:

Execute in WebShell #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.

```
[ ]: . /home/user/.bashrc_ros2
     cd ~/ros2_ws
     # Lista ll the service messages available and filter by name
     ros2 srv list | grep
     unit_3_services_custom_msgs/MyCustomServiceMessage
     # Show contents of message
     ros2 srv show unit_3_services_custom_msgs/MyCustomServiceMessage
```

WebShell #1 Output

```
[ ]: unit_3_services_custom_msgs/MyCustomServiceMessage

     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.

It's also nice to see where the include file is saved because you will need to be sure that it's done if you want to import the messages and use them.

Execute in WebShell #1

```
[ ]: # Check that the gazebo_msgs were compiled for cpp
     ll ~/ros2_ws/install/unit_3_services_custom_msgs/include/unit_3_servic
     es_custom_msgs/srv/my_custom_service_message.hpp
```

WebShell #1 Output

```
[ ]: lrwxrwxrwx 1 ubuntu ubuntu 137 Dec  7 12:34 /home/ubuntu/ros2_ws/insta
     ll/unit_3_services_custom_msgs/include/unit_3_services_custom_msgs/srv
     /my_custom_service_message.hpp -> /home/ubuntu/ros2_ws/build/unit_3_se
     rvices_custom_msgs/rosidl_generator_cpp/unit_3_services_custom_msgs/sr
     v/my_custom_service_message.hpp
```

Great! So, to use it, you just have to add the following include in your cpp file:

```
[ ]: #include
     "unit_3_services_custom_msgs/srv/my_custom_service_message.hpp"
```

Create a service server that uses this custom message: 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 WebShell #1

```
[ ]: cd ~/ros2_ws/src/cpp_unit_3_services
     touch src/custom_service_server.cpp
```

C++ Program {3.3}: custom_service_server.cpp

```
[ ]: #include <inttypes.h>
     #include <memory>
     #include "rclcpp/rclcpp.hpp"
     #include
     "unit_3_services_custom_msgs/srv/my_custom_service_message.hpp"

     using MyCustomServiceMessage =
     unit_3_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;
     }
```

Execute in WebShell #1

```
[ ]: cd ~/ros2_ws/src/cpp_unit_3_services
     touch src/custom_service_client.cpp
```

C++ Program {3.3}: custom_service_client.cpp

```
[ ]: #include <chrono>
     #include <cinttypes>
     #include <iostream>
     #include <memory>
```

```
#include <string>

#include "rclcpp/rclcpp.hpp"
#include
"unit_3_services_custom_msgs/srv/my_custom_service_message.hpp"

using MyCustomServiceMessage =
unit_3_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;
}
```

Now, we make the necessary changes to the **CMakeLists.txt** to compile it. For that, we have to add the custom_executable method call for the new cpp files:

```
[ ]: custom_executable(custom_service_client)
     custom_executable(custom_service_server)
```

And add some dependecies related to **unit_3_services_custom_msgs**:

```
[ ]: find_package(unit_3_services_custom_msgs REQUIRED)
     ...
     add_executable(${target}_node src/${target}.cpp)
       ament_target_dependencies(${target}_node
         "gazebo_msgs"
         "rclcpp"
         "std_msgs"
         "std_srvs"
         "unit_3_services_custom_msgs")
     ...
```

Your CmakeLists.txt should look something like this:

Setup {3.1}: CMakeLists.txt

```
[ ]: cmake_minimum_required(VERSION 3.5)
     project(cpp_unit_3_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_3_services_custom_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()

     function(custom_executable target)
       add_executable(${target}_node src/${target}.cpp)
       ament_target_dependencies(${target}_node
         "gazebo_msgs"
         "rclcpp"
         "std_msgs"
         "std_srvs"
```

```
    "unit_3_services_custom_msgs")
  install(TARGETS ${target}_node
  DESTINATION lib/${PROJECT_NAME})
endfunction()

# Adding Services
custom_executable(cpp_simple_service_client)
custom_executable(cpp_simple_service_dummy_server)
custom_executable(cpp_simple_service_client_ex3_1)
custom_executable(empty_service_server)
custom_executable(custom_service_client)
custom_executable(custom_service_server)

ament_package()

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

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

```
[ ]: <depend>unit_3_services_custom_msgs</depend>
```

Your package.xml should look something like this:

Setup {3.2}: 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_3_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_3_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>
```

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

Execute in WebShell #1:

```
[ ]: . /home/user/.bashrc_ros2
     cd ~/ros2_ws
     # We compile only our cpp_unit_3_services package, nothing more
     colcon build --symlink-install --packages-select cpp_unit_3_services
```

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

Execute in WebShell #2: Launch Service Server

```
[ ]: . /home/user/.bashrc_ros2
     ros2 run cpp_unit_3_services custom_service_server_node
```

Execute in WebShell #3: Start Client

```
[ ]: . /home/user/.bashrc_ros2
     ros2 run cpp_unit_3_services custom_service_client_node
```

WebShell #2 Output

```
[ ]: [INFO] [custom_service_server]: CustomServiceServer...READY
     [INFO] [custom_service_server]: Incoming request
     radius: 2.300000
     [INFO] [custom_service_server]: Incoming request
     repetitions: 2
```

WebShell #3 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 WebShell #2: Call the Server

```
[ ]: . /home/user/.bashrc_ros2
     ros2 service call /my_custom_service
     unit_3_services_custom_msgs/MyCustomServiceMessage '{radius:
     5.1,repetitions: 7}'
```

WebShell #1 Output

```
[ ]: [INFO] [custom_service_server]: Incoming request
     radius: 5.100000
     [INFO] [custom_service_server]: Incoming request
     repetitions: 7
```

Exercise 3.3
Modify the **square_service_server.cpp** and its client that you generated in the previous exercise **3.2** to be able to request different-sized squares and number of repetitions using the **unit_3_services_custom_msgs/MyCustomServiceMessage**.

Fig.3.2 - BB8 Dynamic Square Diagram

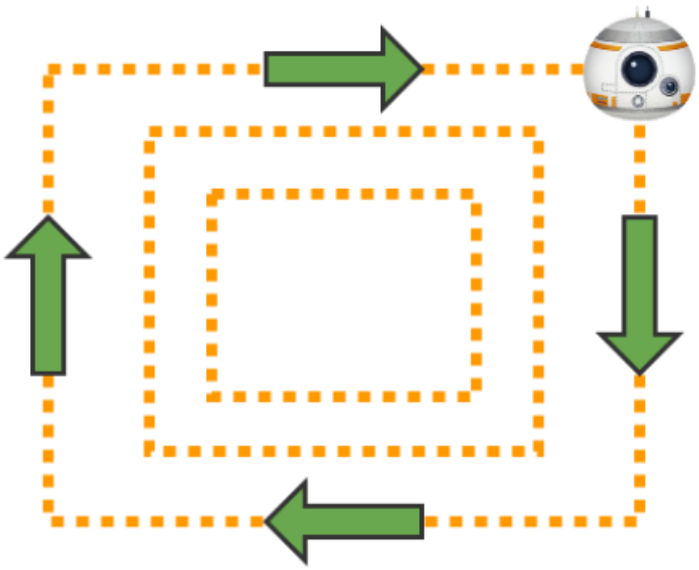
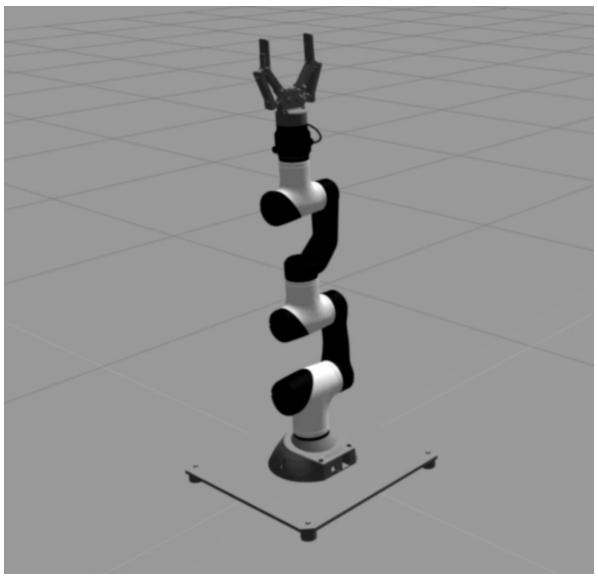


Fig.3.2 - BB8 Dynamic Square Diagram

Unit 5. Debugging Tools

ROS2 BASICS IN 5 DAYS

Unit 5: Debugging Tools



Mara Robot

- ROSject Link: https://bit.ly/2Be6B3x
- Robot: **MARA**

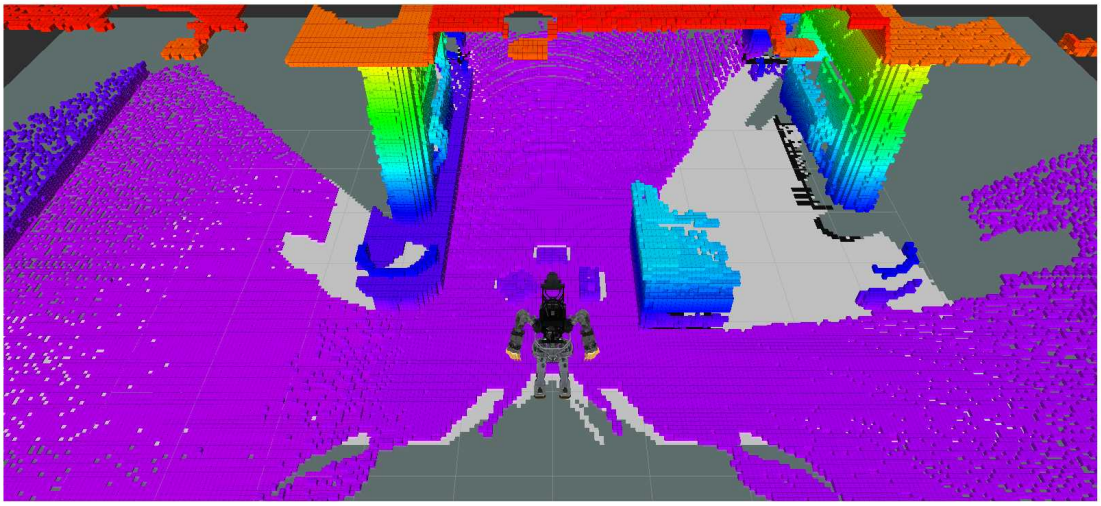
NOTE: You will find instructions on how to launch this simulation in the Jupyter Notebook of the ROSject.

Estimated time to completion: 1.5 hours What will you learn with this unit?

- Add Debugging ROS logs
- Basic use of RViz2 debugging tool

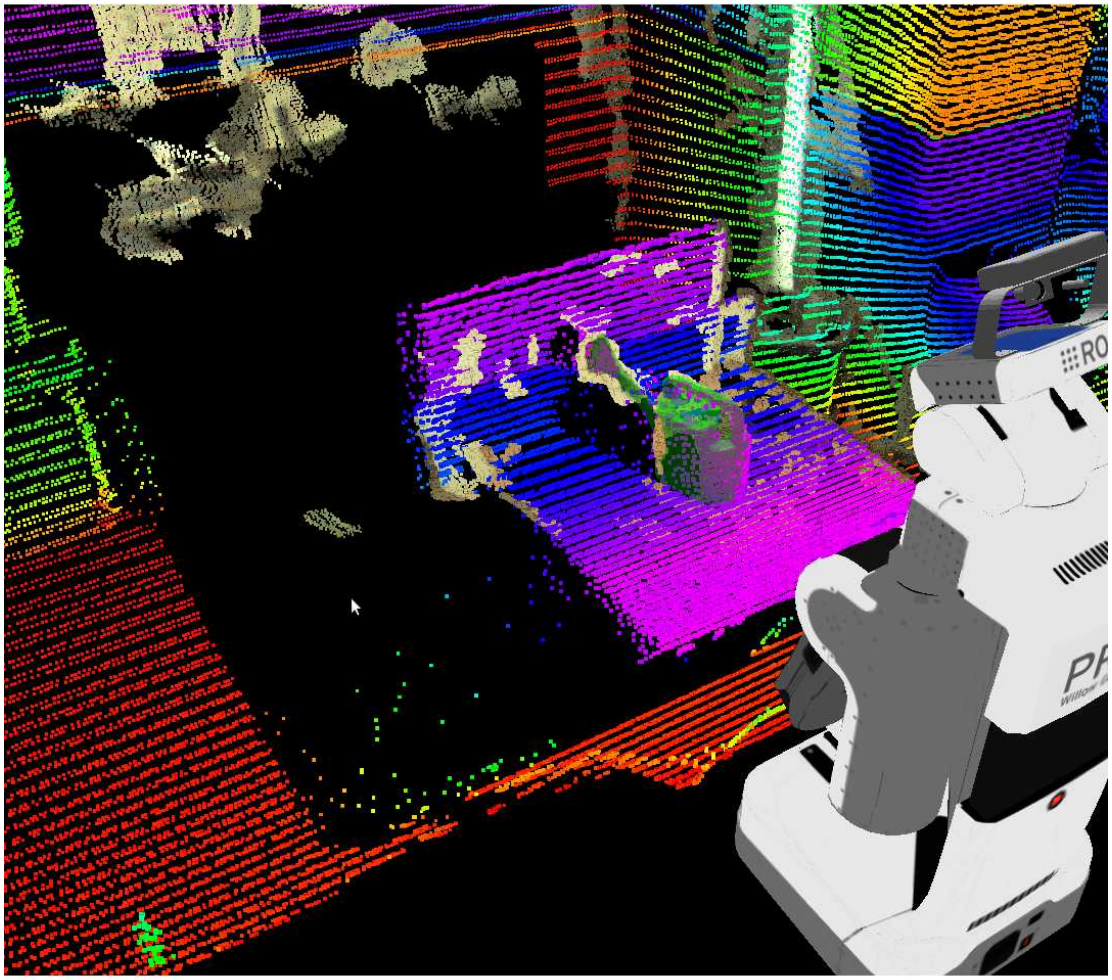
One of the most difficult, but important, parts of robotics is: **knowing how to turn your ideas and knowledge into real projects**. There is a constant in robotics projects: **nothing works as in theory**. Reality is much more complex and, therefore, you need tools to discover what is going on and find 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-5.i} and {Fig-5.ii}.

Fig.5.i - Atlas Laser



Rviz Example 1

Fig.5.ii - PR2 Laser and PointCloud



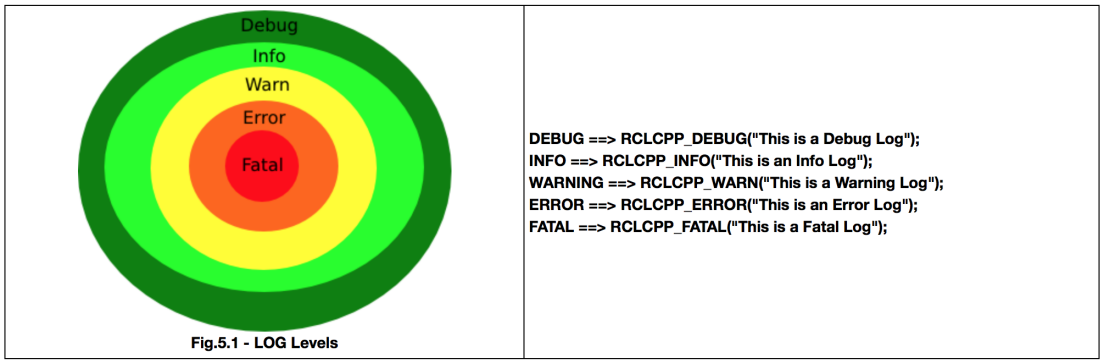
Rviz Example 2

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.

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 {Fig-5.1}. 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.



LOG Levels

Run the following C++ code:

Exercise 5.1

- 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 {5.1}: 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;
}
```

When compiling, you will probably see the following warning:

```
--- stderr: logs_test
/home/user/ros2_ws/src/logs_test/src/logger_example.cpp: In func
tion 'int main(int, char**)':
/home/user/ros2_ws/src/logs_test/src/logger_example.cpp:9:35: wa
rning: ignoring return value of 'rcutils_ret_t rcutils_logging_s
et_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);
   ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~^~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
---
```

Compilation Warning

Just ignore it, it won't affect the exercise.

You should see all of the ROS 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
```

Logs Output

Exercise 5.2

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>);
```

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 WebShell #1

```
[ ]: source /opt/ros/crystal/setup.bash
```

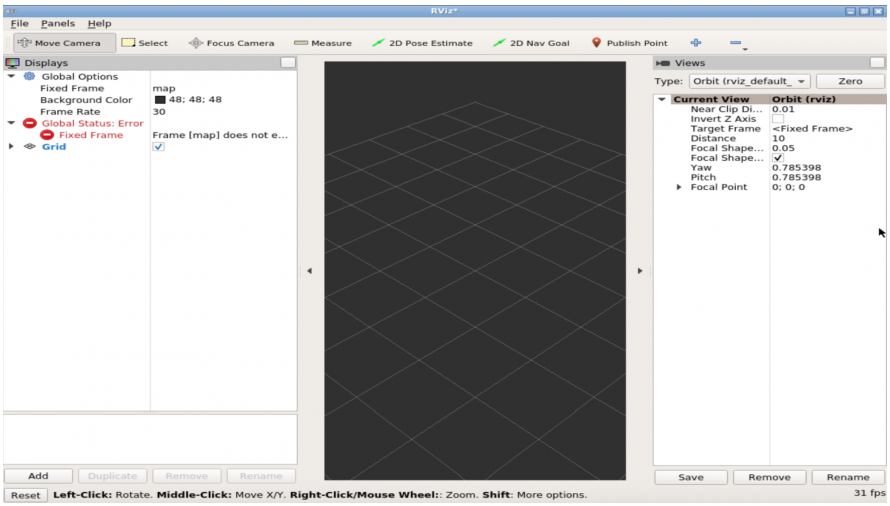
```
[ ]: source /home/simulations/ros2_sims_ws/install/setup.bash
```

```
[ ]: rviz2
```

2- Then, go to the graphical interface to see the RVIZ2 GUI:

You will be greeted by a window like {Fig-5.9}:

Fig-5.9 - RVIZ Starting Window

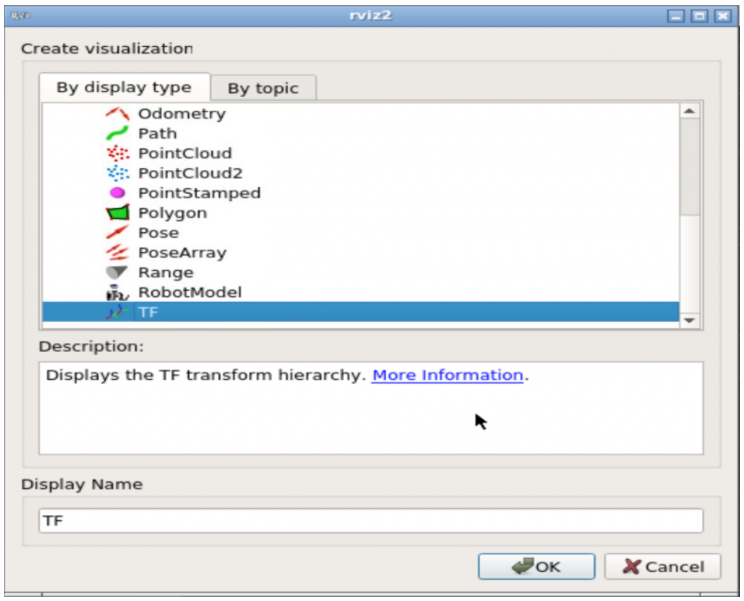


RVIZ Starting Window

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.
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.

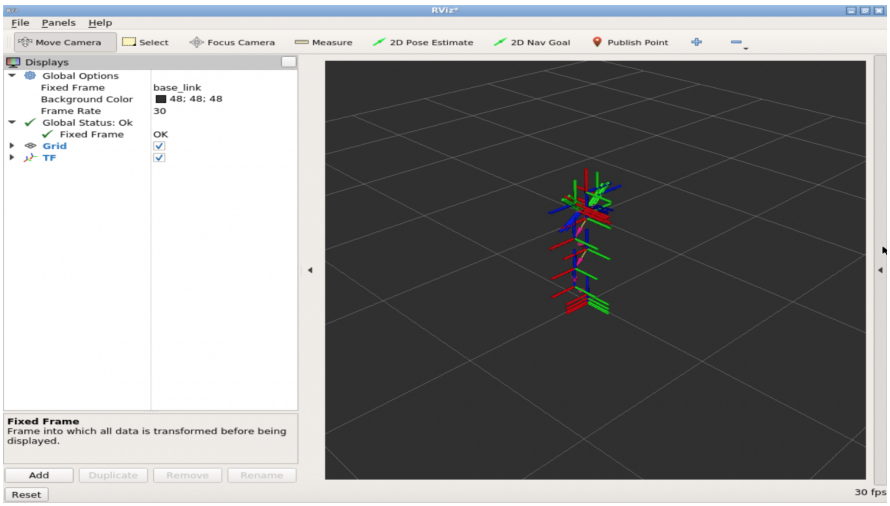
Go to RVIZ in the graphical interface and add a TF element. For that, click "Add" and select the element TF in the list of elements provided, as shown in {Fig-5.10}.
Fig-5.10 - RVIZ Add element



RVIZ Add element

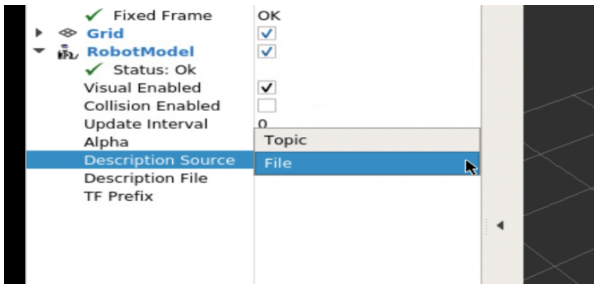
- Go to the RVIZ2 Left panel, select the base_link as Fixed Frame, and make sure that the TF element checkbox is checked. In a few moments, you should see all of the Robot's Elements Axis represented in the CENTRAL Panel.

Fig-5.11 - RVIZ TF's



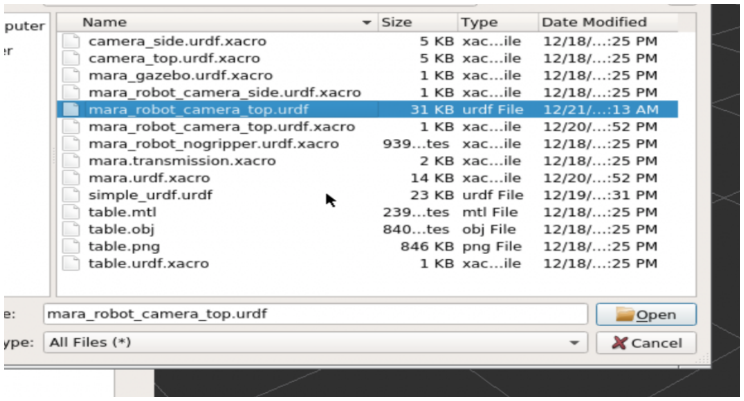
RVIZ TF's

- Now, press "Add" and select RobotModel, as shown in {Fig-5.10}
- On the RobotModel options, set the "Description Source" to **File**.



RVIZ Robot Model Configuration

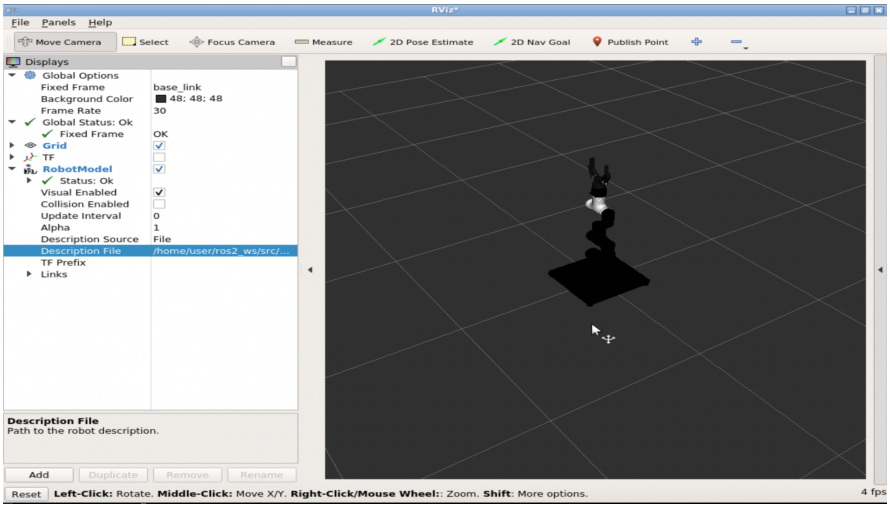
- Finally, select the file named **mara_robot_camera_top.urdf**. You will find it at **/home/simulations/ros2_sims_ws/src/ros2_mara/MARA/mara_description/urdf**.



URDF File Selection

You should now see the 3D model of the robot, as shown in {Fig-5.12}:

Fig-5.12 - RVIZ Robot Model



RVIZ Robot Model

- Now, go to WebShell #2 and enter the command to move the robot:

Execute in WebShell #2

```
[ ]: source /opt/ros/crystal/setup.bash

[ ]: source /home/simulations/ros2_sims_ws/install/setup.bash

[ ]: roslaunch iri_wam_aff_demo start_demo.launch
```

You should see something like this:

Fig-5.13 - RVIZ TF

RVIZ Robot Model + TF

In {Fig-5.11}, you are seeing all of the transformation elements of the IRI Wam Simulation 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

I have finished, now what?

ROS Development Studio (ROSDS)

ROSDS logo

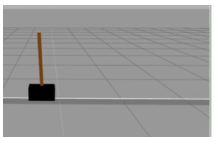
ROSDS is the The Construct web based tool to program ROS robots online. It requires no installation in your computer. Hence, you can use any type of computer to work on it (Windows, Linux or Mac). Additionally, free accounts are available. **Create a free ROSDS account here**: http://rosds.online
You can use any of the many ROSjects available in order to apply all the things you've learned during the Course. You just need to paste the ROSject link to your browser's URL, and you will automatically have the simulation prepared in your ROSDS workspace.

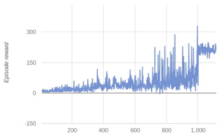
Down below you can check some examples of the Public Rosjects we provide:

ARIAC Competition

- ROSject Link: https://bit.ly/2t2px0t

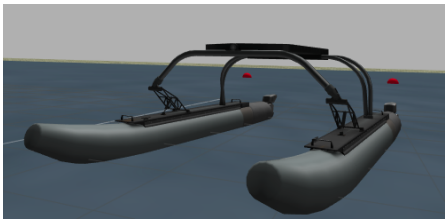
Cartpole Reinforcement Learning





- ROSject Link: https://bit.ly/2t2uGWr

RobotX Challenge



- ROSject Link: https://bit.ly/2Tt4lw8

Want to learn more?

Robot Ignite Logo

Once you have finished the course, you can still learn a lot of interesting ROS subjects.

- Take more advanced courses that we offer at the Robot Ignite Academy, like Perception or Navigation. Access the Academy here: http://www.robotigniteacademy.com
- Or, you can go to the ROS Wiki and check the official documentation of ROS, now with new eyes.

Thank You and hope to see you soon