ROS2 Basics in 5 Days (Python)

Unit 3 Understanding ROS2 Topics

- Summary -

Estimated time to completion: **3 hours**

What will you learn about in this unit?

- Topics
- Basic Topic commands
- Topic Publishers
- Topic Subscribers
- How to mix Publishers and Subscribers
- Custom interfaces

- End of Summary -

3.1 What is a Topic in ROS2?

I know you are eager to start playing with the simulation, but you must be clear on a few concepts before you begin. So let us start with Topics.

3.1.1 Basic Topic commands

Open your Shell #1 and run the commands to source it for working with ROS2.

Execute in Shell #1

In [ ]:

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

Okay. Now you are ready to get to work. Start by trying the following command:

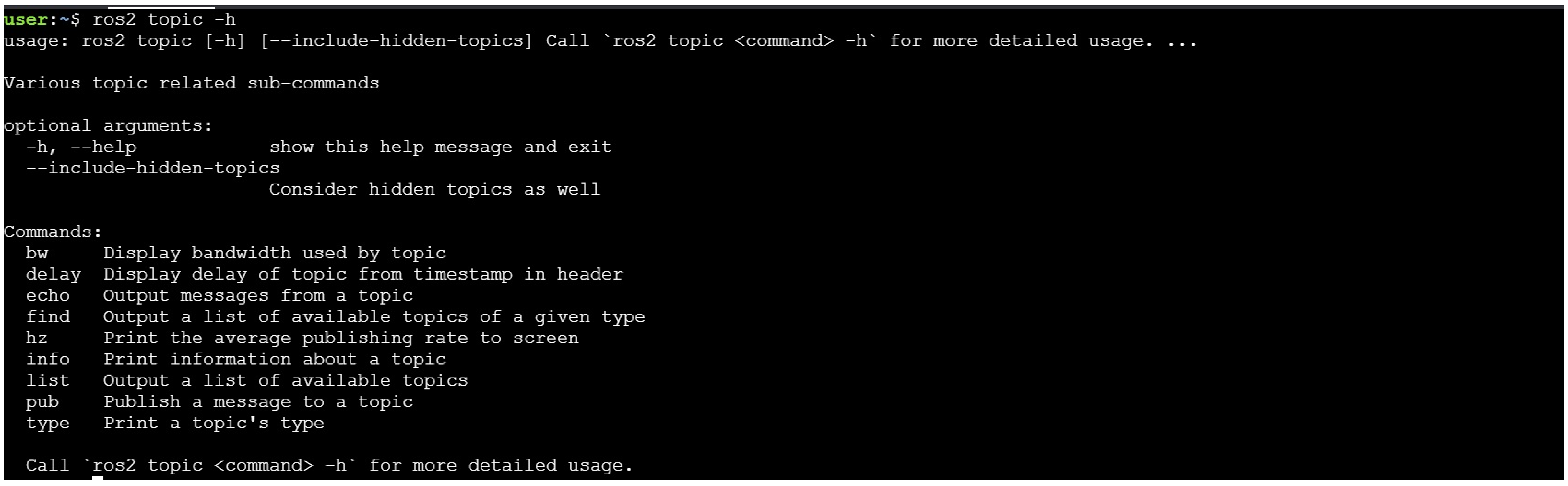
Execute in Shell #1

In [ ]:

```
ros2 topic -h
```

You should get a result equal to the following:

Shell #1 Output



```
user:~$ ros2 topic -h
usage: ros2 topic [-h] [--include-hidden-topics] Call `ros2 topic <command> -h` for more detailed usage. ...

Various topic related sub-commands

optional arguments:
  -h, --help            show this help message and exit
  --include-hidden-topics
                        Consider hidden topics as well

Commands:
  bw     Display bandwidth used by topic
  delay  Display delay of topic from timestamp in header
  echo   Output messages from a topic
  find   Output a list of available topics of a given type
  hz     Print the average publishing rate to screen
  info   Print information about a topic
  list   Output a list of available topics
  pub    Publish a message to a topic
  type   Print a topic's type

  Call `ros2 topic <command> -h` for more detailed usage.
```

The most important results to be reviewed in this section of the course are the following: `echo`, `info`, `list`, `pub`.

- Example 3.1 -

Type the following command in the shell:

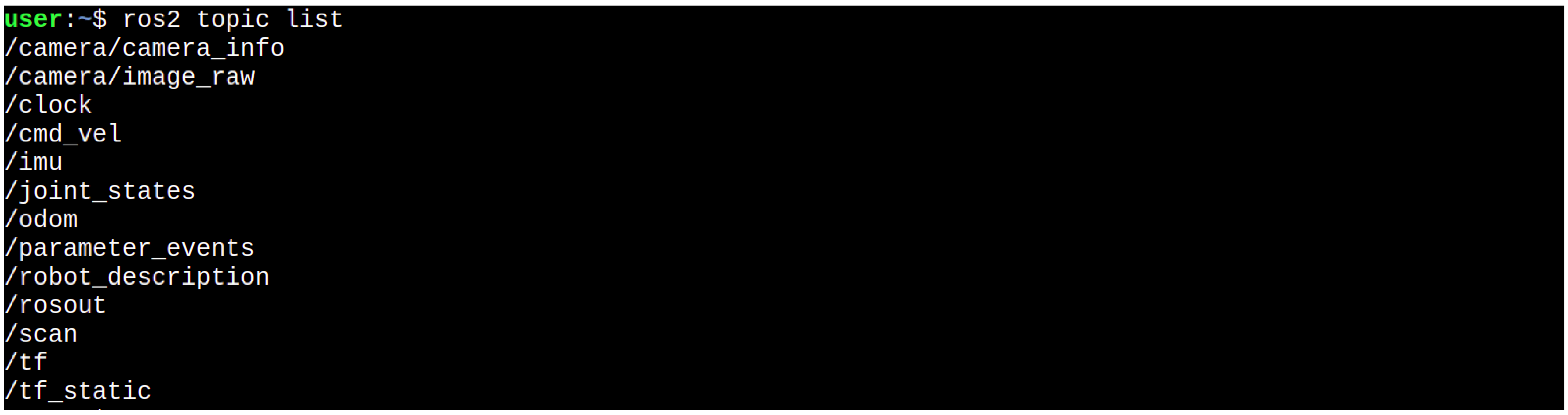
Execute in Shell #1

In [ ]:

```
ros2 topic list
```

You should get a result similar to the image below:

Shell #1 Output



```
user:~$ ros2 topic list
/camera/camera_info
/camera/image_raw
/clock
/cmd_vel
/imu
/joint_states
/odom
/parameter_events
/robot_description
/rosout
/scan
/tf
/tf_static
```

You have listed the Topics in your environment currently available to work on.

- Notes -

The `ros2 topic list` command prints a list of all the available Topics.

- End of Notes -

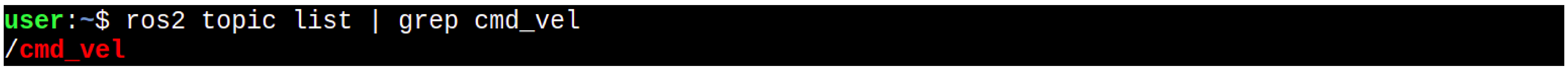
Now try searching for a specific Topic. For this, you need the following command:

Execute in Shell #1

In [ ]:

```
ros2 topic list | grep cmd_vel
```

Shell #1 Output



```
user:~$ ros2 topic list | grep cmd_vel
/cmd_vel
```

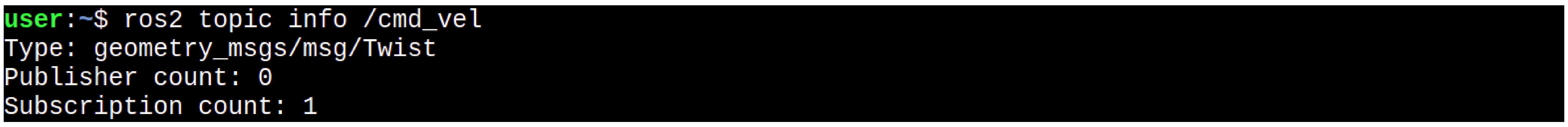
It looks like you have found it! Now, try to get some information about it using the following command:

Execute in Shell #1

In [ ]:

```
ros2 topic info /cmd_vel
```

Shell #1 Output



```
user:~$ ros2 topic info /cmd_vel
Type: geometry_msgs/msg/Twist
Publisher count: 0
Subscription count: 1
```

Okay. Now, break down what you have got a little bit:

- **Type:** Refers to the ROS2 interface associated with the Topic with which you need to work with this Topic.
- **Publisher count:** Refers to the number of active Publishers connected to the Topic.
- **Subscription count:** Refers to the number of active Subscribers connected to the Topic.

- Notes -

Do not worry about understanding Publishers, Subscribers, and Interfaces. Those concepts will be explained later in this unit. For now, think of them as ROS2 agents actively associated with a Topic.

- End of Notes -

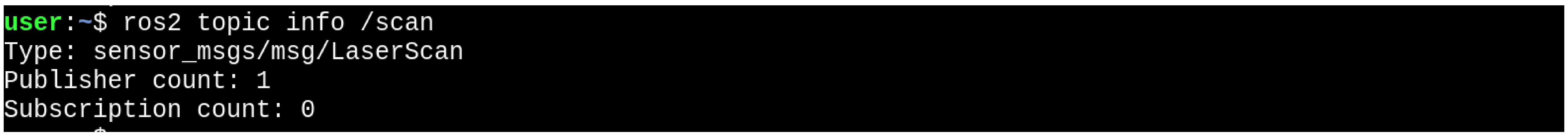
Okay. Now, try running the same command with different Topics to see what differences you find:

Execute in Shell #1

In [ ]:

```
ros2 topic info /scan
```

Shell #1 Output



```
user:~$ ros2 topic info /scan
Type: sensor_msgs/msg/LaserScan
Publisher count: 1
Subscription count: 0
```

Note that there is a count of 1 Publisher active in this Topic, and there is a count of 0 Subscribers active in this Topic. This is the opposite case for the Topic you reviewed previously.

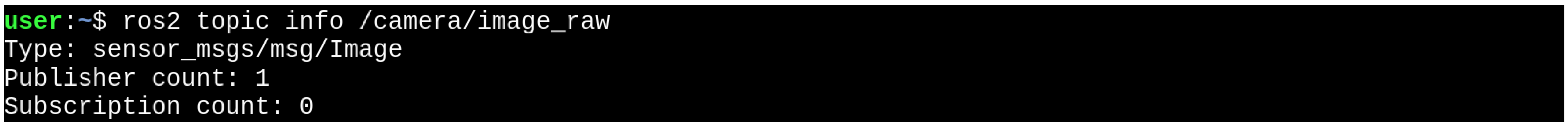
Try one more.

Execute in Shell #1

In [ ]:

```
ros2 topic info /camera/image_raw
```

Shell #1 Output



```
user:~$ ros2 topic info /camera/image_raw
Type: sensor_msgs/msg/Image
Publisher count: 1
Subscription count: 0
```

You likely have obtained a similar result to the previous one, with one publisher and zero subscriptions. It may be because these are the cases of the data from the robot's sensors. The robot is publishing this data, but for now, nobody is subscribed to it.

In this case, the `cmd_vel` Topic is different because it represents an actuator; in this case, the robot motors. The robot is subscribed to this Topic because it waits for someone to publish velocity commands in it.

- Notes -

I invite you to try to get information from the other Topics present in your work environment. You might find something interesting.

- End of Notes -

- Notes -

The **`ros2 topic info`** command is used to get information about a specific Topic. The command structure is as follows:

In [ ]:

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

- End of Notes -

- End of Example 3.1 -

So far, so good. You have learned to list the Topics present in your work environment. You have learned to look for a specific Topic and find information about it. Now let us dig deeper into how to work with the Topics.

- Example 3.2 -

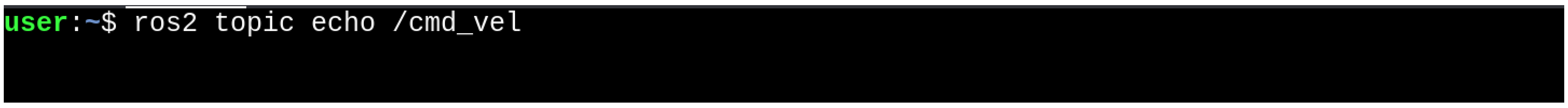
Look at how you can check the current data for a specific Topic. To do this, run the following command in the same shell:

Execute in Shell #1

In [ ]:

```
ros2 topic echo /cmd_vel
```

Shell #1 Output



```
user:~$ ros2 topic echo /cmd_vel
```

- Notes -

The **`ros2 topic echo`** command is used to read (subscribe to) the published Topic. The command structure is as follows:

In [ ]:

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

- End of Notes -

Strange. There is absolutely nothing wrong.

Okay. Do not panic. There is nothing wrong with the command you just typed. There is nothing broken in your workspace, either. What happens here is that at this moment, nobody is publishing any data to the Topic `/cmd_vel`. As you saw before, it is a Topic with a subscription count equal to one. Currently, only the TurtleBot3 robot is subscribed to this Topic. So, try publishing something in this Topic.

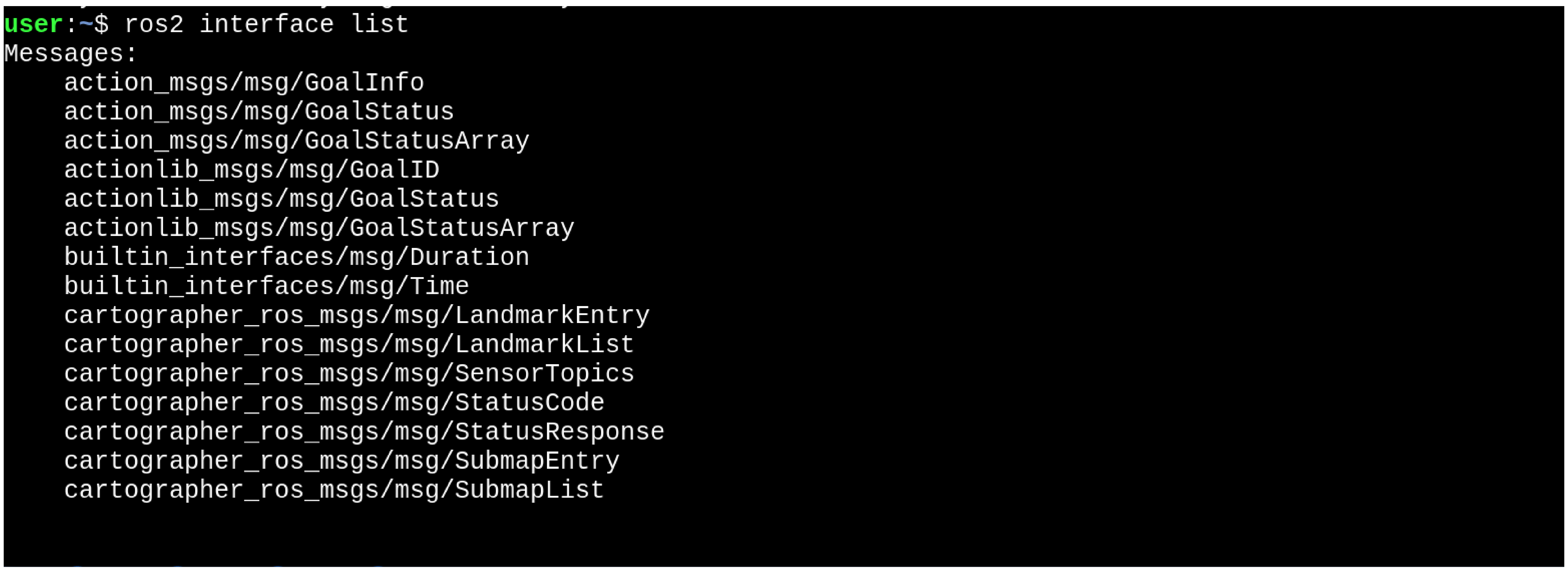
To try publishing something in this Topic, you have to **first check what type of interface the Topic uses**. Then, try a new command named `interface`. As you did before, start by listing the available interfaces. To do this, type the following command in your Shell #1:

Execute in Shell #1

In [ ]:

```
ros2 interface list
```

Shell #1 Output



```
user:~$ ros2 interface list
Messages:
    action_msgs/msg/GoalInfo
    action_msgs/msg/GoalStatus
    action_msgs/msg/GoalStatusArray
    actionlib_msgs/msg/GoalID
    actionlib_msgs/msg/GoalStatus
    actionlib_msgs/msg/GoalStatusArray
    builtin_interfaces/msg/Duration
    builtin_interfaces/msg/Time
    cartographer_ros_msgs/msg/LandmarkEntry
    cartographer_ros_msgs/msg/LandmarkList
    cartographer_ros_msgs/msg/SensorTopics
    cartographer_ros_msgs/msg/StatusCode
    cartographer_ros_msgs/msg/StatusResponse
    cartographer_ros_msgs/msg/SubmapEntry
    cartographer_ros_msgs/msg/SubmapList
```

- Notes -

The **`ros2 interface list`** command prints a list of all the available interfaces.

- End of Notes -

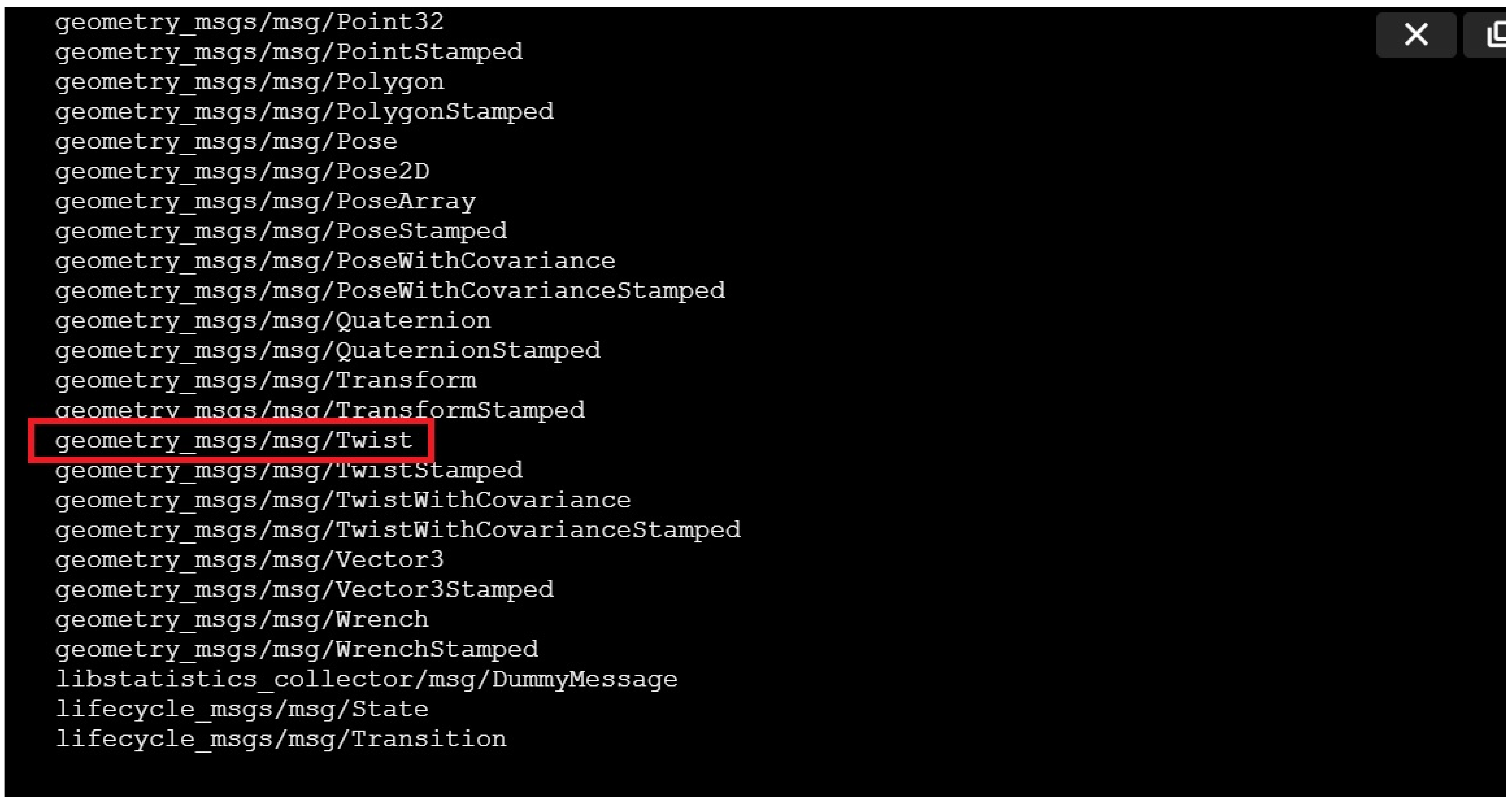
You should find a similar result in the terminal where you ran the command. Looking at what the shell shows you, you will see that the interfaces are divided into the following groups:

- **Messages:** Can be found as `.msg` files. They are simple text files that describe the fields of a ROS message. You can use them to generate source code for messages.
- **Services:** Can be found as `.srv` files. They are composed of two parts: a request and a response. Both are message declarations.
- **Actions:** Can be found as `.action` files. They are composed of three parts: a goal, a result, and feedback. Each part contains a message declaration.

You are now interested in messages, so let us go a little deeper into what they are.

Okay. There are two ways to find the descriptions and definitions for messages: in the `.msg` files and the `msg/` directory of a ROS package. The `.msg` files are composed of two parts: **fields** and **constants**. You can find more information about this in the official ROS 2 documentation.

Resume what you were doing with the `cmd_vel` Topic. Then, looking at the list of interfaces you displayed, you will find the following:



```
geometry_msgs/msg/Point32
geometry_msgs/msg/PointStamped
geometry_msgs/msg/Polygon
geometry_msgs/msg/PolygonStamped
geometry_msgs/msg/Pose
geometry_msgs/msg/Pose2D
geometry_msgs/msg/PoseArray
geometry_msgs/msg/PoseStamped
geometry_msgs/msg/PoseWithCovariance
geometry_msgs/msg/PoseWithCovarianceStamped
geometry_msgs/msg/Quaternion
geometry_msgs/msg/QuaternionStamped
geometry_msgs/msg/Transform
geometry_msgs/msg/TransformStamped
geometry_msgs/msg/Twist
geometry_msgs/msg/TwistStamped
geometry_msgs/msg/TwistWithCovariance
geometry_msgs/msg/TwistWithCovarianceStamped
geometry_msgs/msg/Vector3
geometry_msgs/msg/Vector3Stamped
geometry_msgs/msg/Wrench
geometry_msgs/msg/WrenchStamped
libstatistics_collector/msg/DummyMessage
lifecycle_msgs/msg/State
lifecycle_msgs/msg/Transition
```

As you may remember, when you were looking for information about the `/cmd_vel` Topic, you were shown that this Topic uses a type called `geometry_msgs/Twist`. As you now know, it is a message-type interface.

Now get some data about this specific type. Execute the following command in Shell #1:

Execute in Shell #1

In [ ]:

```
ros2 interface show geometry_msgs/msg/Twist
```

Shell #1 Output

```
# This expresses velocity in free space broken into its linear and angular parts.

Vector3  linear
        float64 x
        float64 y
        float64 z
Vector3  angular
        float64 x
        float64 y
        float64 z
```

What an exciting thing you just discovered! The last command output tells you that the `Twist` message comprises two `Vector3` messages. It consists of a 3-dimensional vector for the linear velocity and another 3-dimensional vector for the angular velocity.

- Notes -

The **`ros2 interface show`** command is used to get information about a specific interface. The command structure is as follows:

In [ ]:

```
ros2 interface show <interface_name>
```

- End of Notes -

- Notes -

Try typing `ros2 interface -h` in the shell then you will know what else can be done with this command. For the moment, I am only showing you the most necessary, so that it can be easily understood, but I invite you to investigate more on your own.

- End of Notes -

Following the Notes section above, you probably found a command called `ros2 interface proto`. This command shows you a prototype interface so that you can use it. To know the structure of the Topic that you were investigating, `/cmd_vel`, execute the following line in your Shell #1:

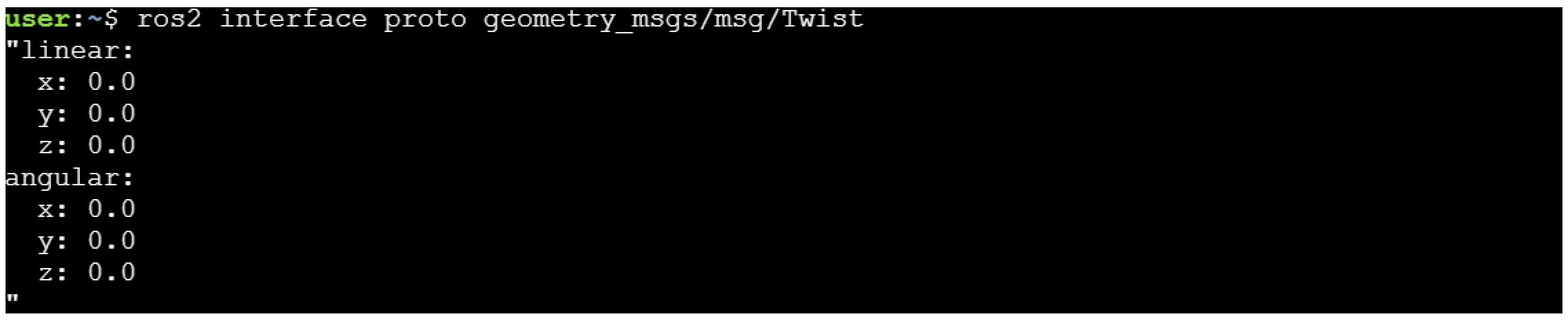
Execute in Shell #1

In [ ]:

```
ros2 interface proto geometry_msgs/msg/Twist
```

You should get a result similar to the following in your shell:

Shell #1 Output



```
user:~$ ros2 interface proto geometry_msgs/msg/Twist
"linear:
  x: 0.0
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 0.0
"
```

Analyze what you have obtained by executing this last command. You can see that the structure to write a message in the Topic `/cmd_vel` is composed of two three-dimensional vectors. One vector is called linear with positions for the axes (x, y, z), and another is called angular with positions for the axes (x, y, z). Recall that using the `ros2 interface show` command, you verified that the Twist message is composed of two messages of type `Vector3`.

You will use the `/cmd_vel` Topic to send a message composed of these velocity vectors to move the TurtleBot robot.

All right. You have enough information to start publishing messages to the `cmd_vel` Topic. To do this, you will use the `geometry_msgs/msg/Twist` interface.

To publish a message in a particular Topic, use the following command:

Execute in Shell #1

In [ ]:

```
ros2 topic pub /cmd_vel geometry_msgs/msg/Twist "{linear: {x: 0.1, y: 0.0, z: 0.0}, angular: {x: 0.0, y: 0.0, z: 1.0}}"
```

- Notes -

The **`ros2 topic pub`** command is used to publish messages to a Topic. The command structure is as follows:

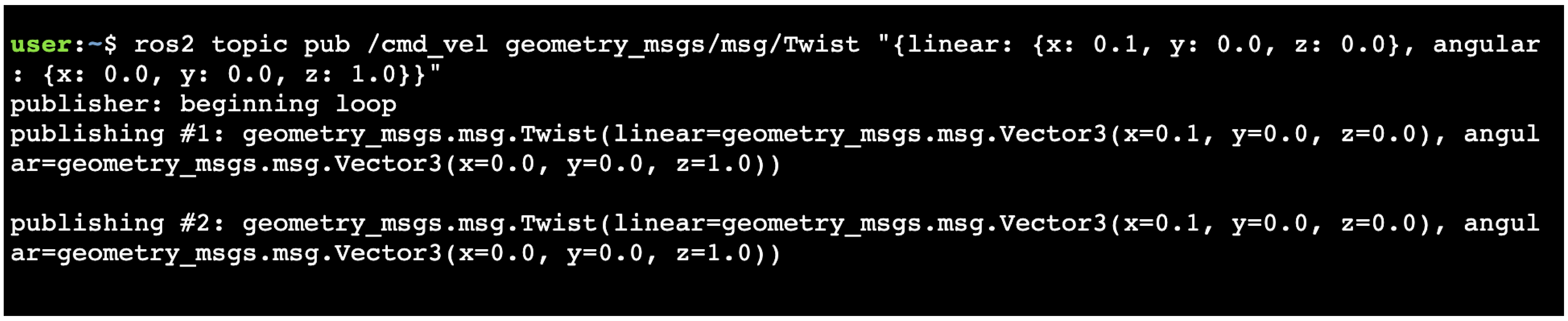
In [ ]:

```
ros2 topic pub <topic_name> <interface_name> <message>
```

- End of Notes -

You should be able to see a message similar to the following on your terminal:

Shell #1 Output



```
user:~$ ros2 topic pub /cmd_vel geometry_msgs/msg/Twist "{linear: {x: 0.1, y: 0.0, z: 0.0}, angular
: {x: 0.0, y: 0.0, z: 1.0}}"
publisher: beginning loop
publishing #1: geometry_msgs.msg.Twist(linear=geometry_msgs.msg.Vector3(x=0.1, y=0.0, z=0.0), angul
ar=geometry_msgs.msg.Vector3(x=0.0, y=0.0, z=1.0))

publishing #2: geometry_msgs.msg.Twist(linear=geometry_msgs.msg.Vector3(x=0.1, y=0.0, z=0.0), angul
ar=geometry_msgs.msg.Vector3(x=0.0, y=0.0, z=1.0))
```

Great! You were able to publish a message to the `/cmd_vel` Topic. Now, see if anything has changed by running the `ros2 topic echo /cmd_vel` command. Try it out. I will be waiting for you here.

Run the following commands in Shell #2:

Execute in Shell #2

In [ ]:

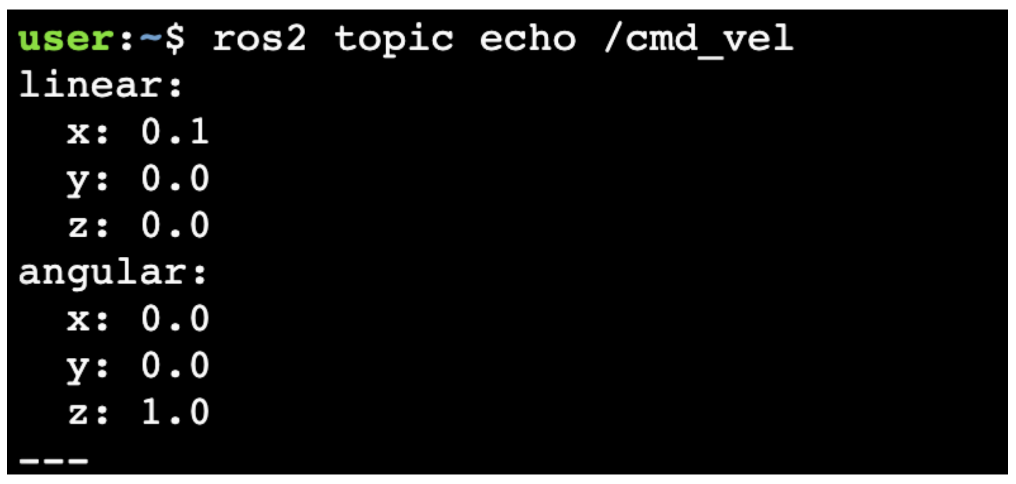
```
source /opt/ros/humble/setup.bash
```

With the shell ready to work, run the following command:

In [ ]:

```
ros2 topic echo /cmd_vel
```

Shell #2 Output



```
user:~$ ros2 topic echo /cmd_vel
linear:
  x: 0.1
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 1.0
---
```

You can also check if the publisher/subscriber count for the `/cmd_vel` Topic has increased.

Execute in Shell #3

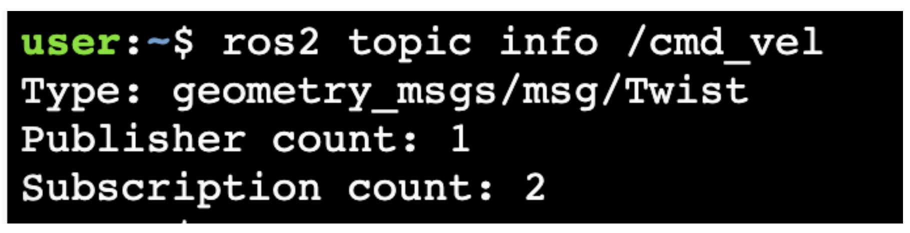
In [ ]:

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

In [ ]:

```
ros2 topic info /cmd_vel
```

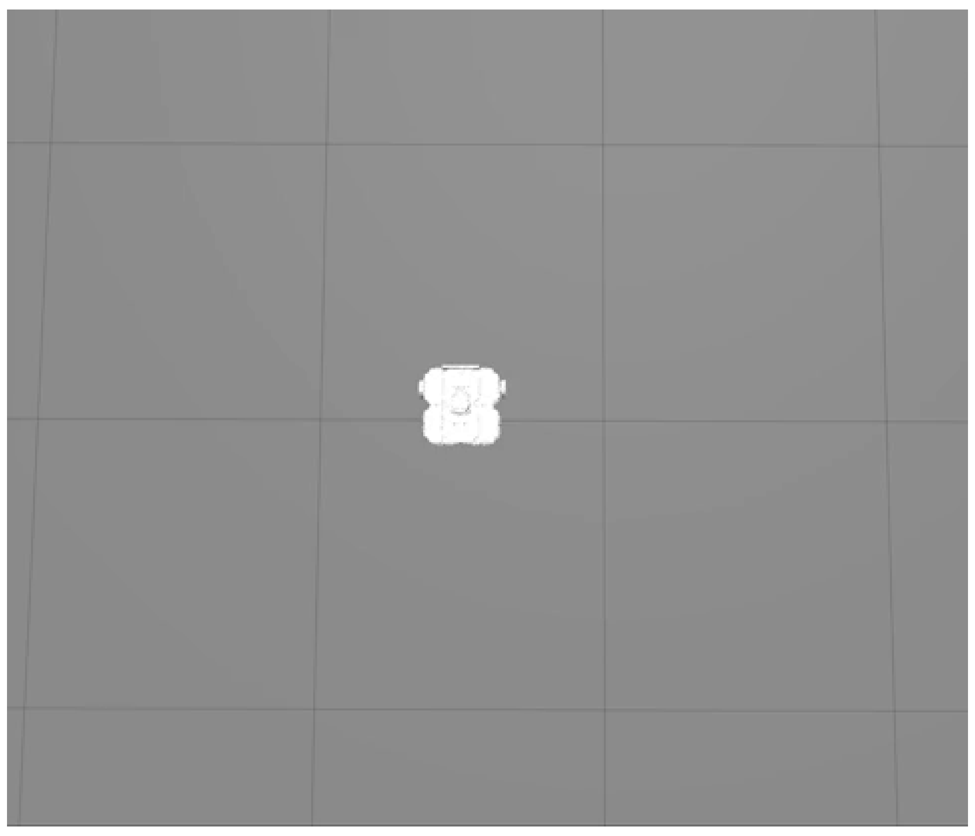
Shell #3 Output



```
user:~$ ros2 topic info /cmd_vel
Type: geometry_msgs/msg/Twist
Publisher count: 1
Subscription count: 2
```

So you can confirm that you have correctly published to the Topic `/cmd_vel`. As you can see, the Publisher count is now 1, since you are publishing into the Topic with the **`ros2 topic pub`** command. Also, the Subscription count has increased to 2, because now you are also subscribed to the Topic with the **`ros2 topic echo`** command.

However, this is not all! You probably have already noticed that, in the Gazebo simulation, the robot has started moving! So you should have something similar to the animated image below:



Great! Now you can stop the running commands in **`Shell #1`** and **`Shell #2`** by pressing **`Ctrl + C`** in the shells.

You will notice that despite stopping the command, the robot still moves. To stop the robot, use the following command:

Execute in Shell #1

In [ ]:

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

- End of Example 3.2 -

- Notes -

There is no problem if you want to restart your simulation. For the following example, stop executing the commands you left running during **Example 3.2**.

- End of Notes -

You have made the TurtleBot3 simulation work using only simple commands. Now, review more examples to strengthen your knowledge of Topics.

- Example 3.3 -

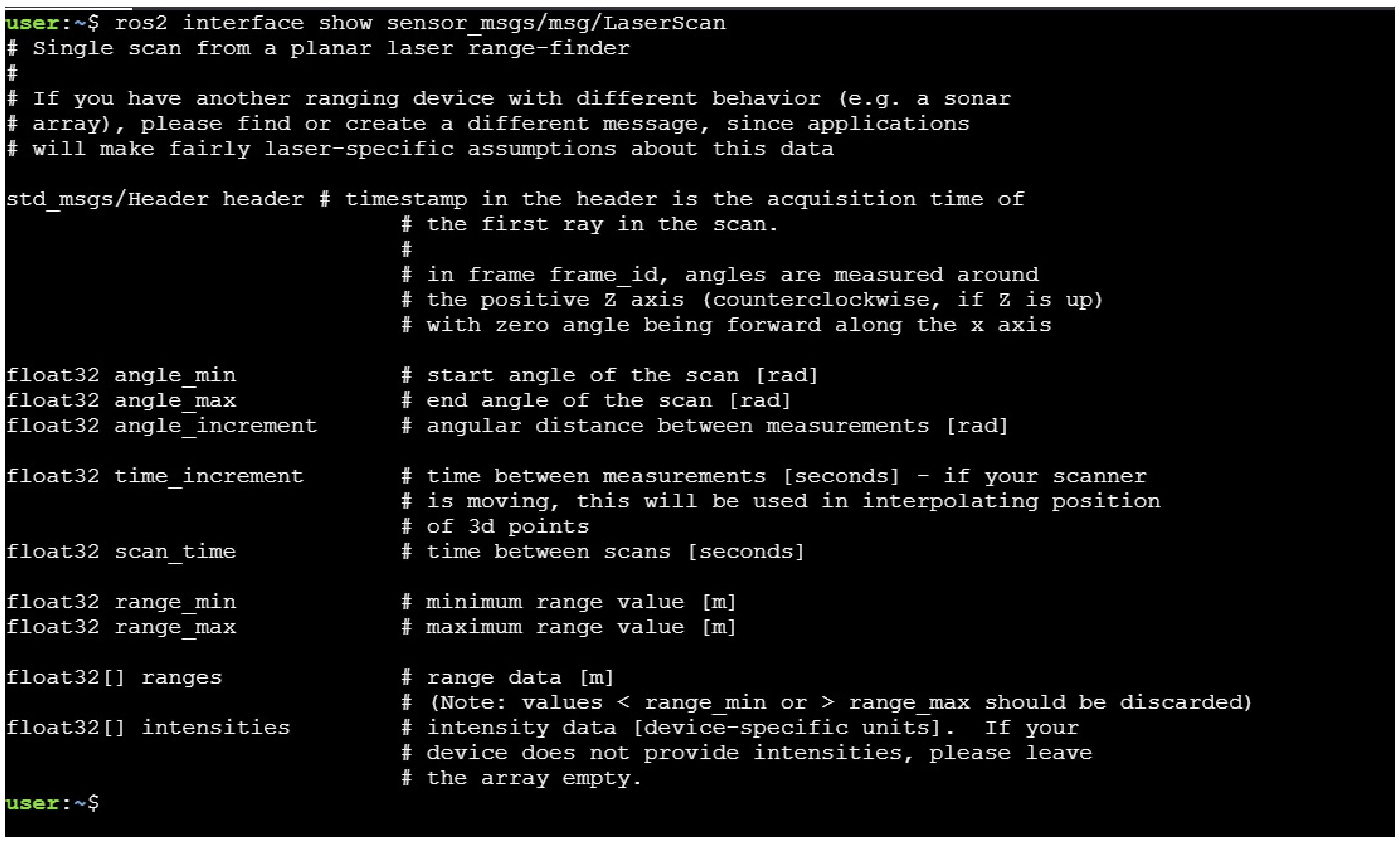
Review how to find important information in other topics, such as the `/scan` Topic. As you have seen before, this Topic works with the `sensor_msgs/msg/LaserScan` interface of type message. Use one of the commands you saw before to extract some information:

Execute in Shell #1

In [ ]:

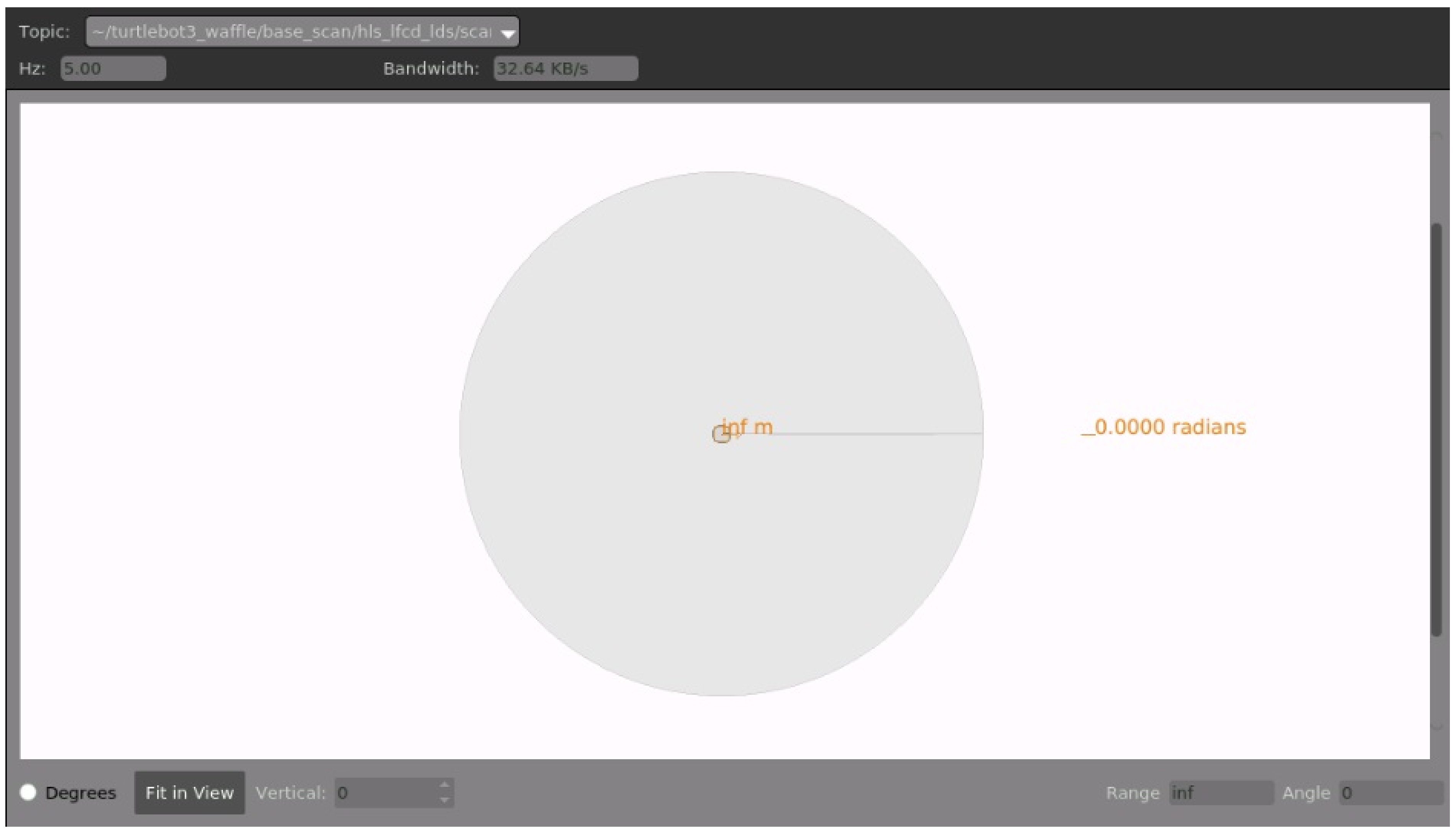
```
ros2 interface show sensor_msgs/msg/LaserScan
```

Shell #1 Output



```
user:~$ ros2 interface show sensor_msgs/msg/LaserScan
# Single scan from a planar laser range-finder
#
# If you have another ranging device with different behavior (e.g. a sonar
# array), please find or create a different message, since applications
# will make fairly laser-specific assumptions about this data

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

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

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

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

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

The terminal shows you that this is the structure of a `LaserScan` message. Complicated?. Let us break it down a little. The Topic `/scan`, with the interface `sensor_msgs/msg/LaserScan`, is used to obtain information about the robot environment. It is a sensor. In this case, a laser sensor returns the distances when the objects close to your robot are within a specific range. The laser scan you are dealing with covers an area of 360º, representing a circular region around the TurtleBot3, as seen in the image below.



As seen in the image above, the area covered by this sensor is circular, works with angles in radians, and returns a distance in meters in a given direction. If the sensor detects no obstacle, it will return an `inf` value.

Now, see how you can learn more about this Topic. But, first, ensure that your shell is ready to work with ROS2.

Execute the following command in your shell:

Execute in Shell #1

In [ ]:

```
ros2 topic echo /scan
```

Your terminal may be flooded with a lot of messages. Remember, you can press **Ctrl+C** to stop running the command. Once you do that, look for a message similar to the following:

Shell #1 Output

```
user:~$ ros2 topic echo /scan
header:
  stamp:
    sec: 499
    nanosec: 403000000
  frame_id: base_scan
angle_min: 0.0
angle_max: 6.28000020980835
angle_increment: 0.01749303564429283
time_increment: 0.0
scan_time: 0.0
range_min: 0.11999999731779099
range_max: 3.5
ranges:
- .inf
- .inf
- .inf
- .inf
- .inf
- .inf
- .inf
- .inf
- .inf
- .inf
```

Take note of the most critical information:

- `angle_min` : Refers to the minimum angle at which the laser works. It can also be seen as the start angle of the scan. [0 rad].
- `angle_max` : Refers to the maximum angle at which the laser works. It can also be seen as the end angle of the scan [2pi rads].
- `angle_increment` : Refers to the angular distance between measurements [rads].

This information is useful because it will help you calculate the indices for each sensor position. For example, for this sensor, you can calculate 360 available positions by dividing the **maximum angle** by the **incremental angle**.

Also, you have:

- `range_min` : Minimum range of the laser [meters].
- `range_max` : Maximum range of the laser [meters].
- `ranges` : An array containing laser detections in all 360 directions [meters].

Each `range` array value represents the sensor's reading in a specific direction. So, each direction is represented as a position in this array. In this case, you have an array with 360 values, where the direction 0° will correspond to position 0, and the direction 45° to position 45. You should always check the sensor's information you want to work with, as the positions and directions will not always correspond as in the case of the sensor you are working with now.

- End of Example 3.3 -

3.1.2 Define a Topic

In the last section, you learned the basics of working with Topics in ROS2, how to list them, find the most important information to use, and how to publish a message and capture it.

Let us look at this. You can send messages to a Topic (known as publishing to a Topic), and read information from a Topic (known as subscribing to it). You can think of it as a channel through which information flows. In Example 3.2, you **published** Twist messages (which represent velocities) into the `/cmd_vel` Topic. Then the robot, subscribed to this Topic, received these messages, and moved accordingly.

Imagine a Topic as a pipe through which information flows, a channel where nodes can read or write information. Finally, you are ready to continue your ROS2 learning path with what you have learned.

3.2 Topic Publisher

A Topic is a channel where information can be written or read. You can now define a **Publisher** as a node that writes information into a Topic.

3.2.1 Create a Simple Publisher Node

- Example 3.4 -

1. Create a new package named **publisher_pkg** in your `~/ros2_ws/src` directory and add the necessary dependencies for working with Topics.

Execute in Shell #1

In [ ]:

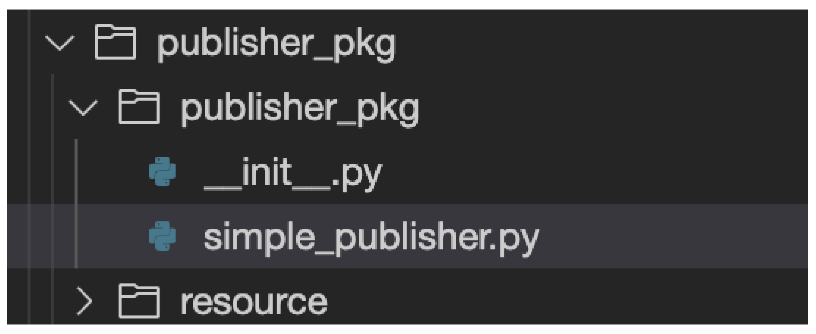
```
ros2 pkg create --build-type ament_python publisher_pkg --dependencies rclpy std_msgs geometry_msgs
```

This should look familiar to you because it is nothing you have not done before. The only new thing is the last two dependencies added, `std_msgs` and `geometry_msgs`. These two packages contain interfaces you will use in your new node.

2. Create a new file named **simple_publisher.py** inside the **publisher_pkg** folder of the package you just created.

- Notes -

Remember to create your scripts inside the `publisher_pkg/publisher_pkg` folder.



- End of Notes -

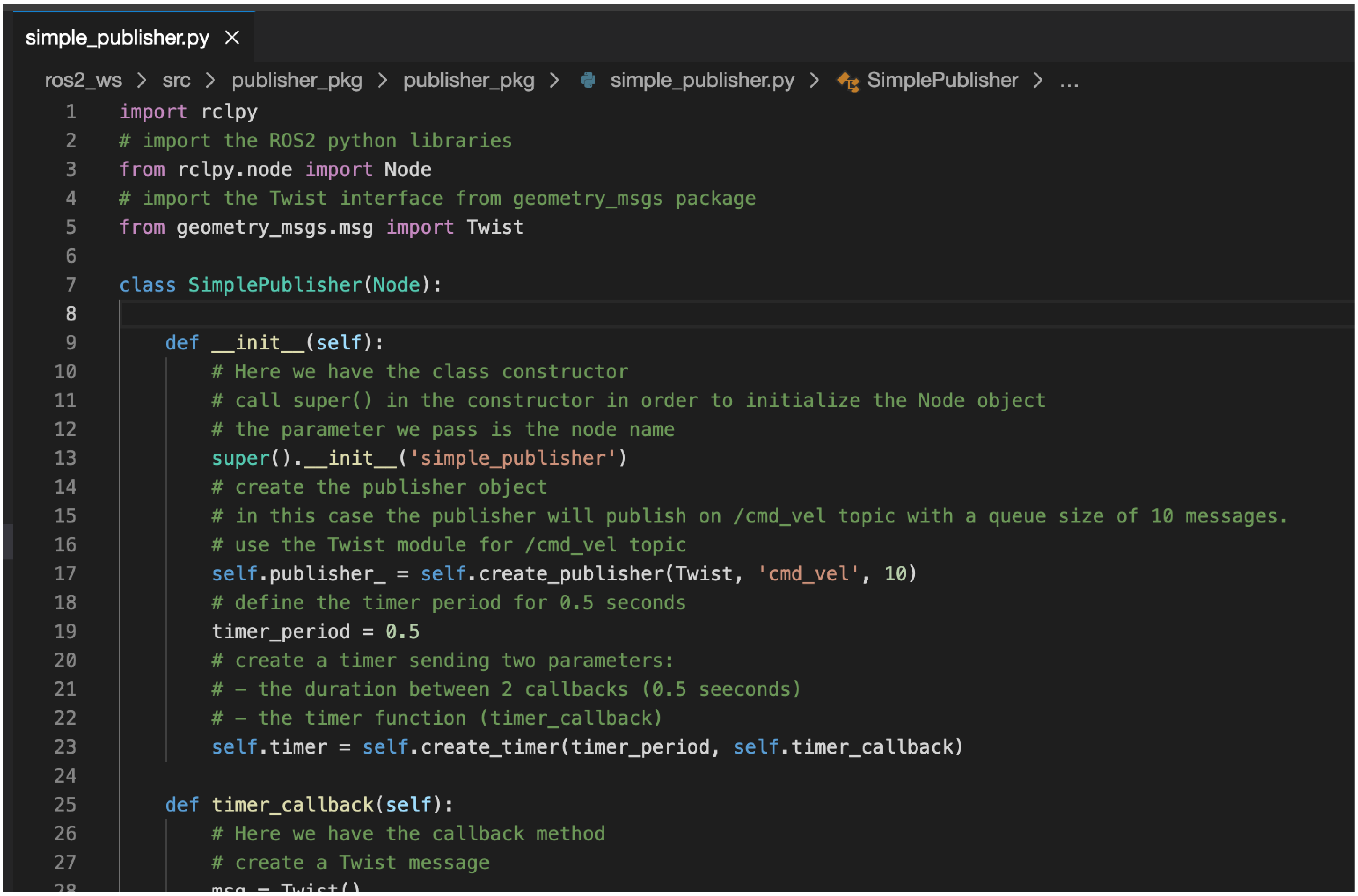
In the file you have just created, copy the following code:

simple_publisher.py

In [ ]:

```python
import rclpy
# import the ROS2 python libraries
from rclpy.node import Node
# import the Twist interface from the geometry_msgs package
from geometry_msgs.msg import Twist

class SimplePublisher(Node):

    def __init__(self):
        # Here you have the class constructor
        # call super() in the constructor to initialize the Node object
        # the parameter you pass is the node name
        super().__init__('simple_publisher')
        # create the publisher object
        # in this case, the publisher will publish on /cmd_vel Topic with a queue size of 10 messages.
        # use the Twist module for /cmd_vel Topic
        self.publisher_ = self.create_publisher(Twist, 'cmd_vel', 10)
        # define the timer period for 0.5 seconds
        timer_period = 0.5
        # create a timer sending two parameters:
        # - the duration between 2 callbacks (0.5 seconds)
        # - the timer function (timer_callback)
        self.timer = self.create_timer(timer_period, self.timer_callback)

    def timer_callback(self):
        # Here you have the callback method
        # create a Twist message
        msg = Twist()
        # define the linear x-axis velocity of /cmd_vel Topic parameter to 0.5
        msg.linear.x = 0.5
        # define the angular z-axis velocity of /cmd_vel Topic parameter to 0.5
        msg.angular.z = 0.5
        # Publish the message to the Topic
        self.publisher_.publish(msg)
        # Display the message on the console
        self.get_logger().info('Publishing: "%s"' % msg)

def main(args=None):
    # initialize the ROS communication
    rclpy.init(args=args)
    # declare the node constructor
    simple_publisher = SimplePublisher()
    # pause the program execution, waits for a request to kill the node (ctrl+c)
    rclpy.spin(simple_publisher)
    # Explicity destroys the node
    simple_publisher.destroy_node()
    # shutdown the ROS communication
    rclpy.shutdown()

if __name__ == '__main__':
    main()
```

In the end, you should have something like this in your code editor:



simple_publisher.py

ros2_ws > src > publisher_pkg > publisher_pkg > simple_publisher.py > SimplePublisher > ...

```python
import rclpy
# import the ROS2 python libraries
from rclpy.node import Node
# import the Twist interface from geometry_msgs package
from geometry_msgs.msg import Twist

class SimplePublisher(Node):

    def __init__(self):
        # Here we have the class constructor
        # call super() in the constructor in order to initialize the Node object
        # the parameter we pass is the node name
        super().__init__('simple_publisher')
        # create the publisher object
        # in this case the publisher will publish on /cmd_vel topic with a queue size of 10 messages.
        # use the Twist module for /cmd_vel topic
        self.publisher_ = self.create_publisher(Twist, 'cmd_vel', 10)
        # define the timer period for 0.5 seconds
        timer_period = 0.5
        # create a timer sending two parameters:
        # - the duration between 2 callbacks (0.5 seeconds)
        # - the timer function (timer_callback)
        self.timer = self.create_timer(timer_period, self.timer_callback)

    def timer_callback(self):
        # Here we have the callback method
        # create a Twist message
```

3. Create a launch file named **publisher_pkg_launch_file.launch.py** to launch the node you have just created.

To do that, execute the following commands:

Execute in Shell #1

In [ ]:
```
cd ~/ros2_ws/src/publisher_pkg
```

In [ ]:
```
mkdir launch
```

Execute in Shell #1

In [ ]:
```
cd ~/ros2_ws/src/publisher_pkg/launch
```

In [ ]:
```
touch publisher_pkg_launch_file.launch.py
```

In [ ]:
```
chmod +x publisher_pkg_launch_file.launch.py
```

Inside the just created new launch file, write the necessary code to launch the executable files of the **simple_publisher** script.

publisher_pkg_launch_file.launch.py

In [ ]:
```python
from launch import LaunchDescription
from launch_ros.actions import Node

def generate_launch_description():
    return LaunchDescription([
        Node(
            package='publisher_pkg',
            executable='simple_publisher',
            output='screen'),
    ])
```

4. Modify the setup.py to install the launch file that you have just created and add the entry point to the executable for the **simple_publisher.py** script.

setup.py

In [ ]:

```
from setuptools import setup
import os
from glob import glob

package_name = 'publisher_pkg'

setup(
    name='publisher_pkg',
    version='0.0.0',
    packages=[package_name],
    data_files=[
        ('share/ament_index/resource_index/packages',
            ['resource/' + package_name]),
        ('share/' + package_name, ['package.xml']),
        (os.path.join('share', package_name), glob('launch/*.launch.py'))
    ],
    install_requires=['setuptools'],
    zip_safe=True,
    maintainer='user',
    maintainer_email='user@todo.todo',
    description='TODO: Package description',
    license='TODO: License declaration',
    tests_require=['pytest'],
    entry_points={
        'console_scripts': [
            'simple_publisher = publisher_pkg.simple_publisher:main'
        ],
    },
)
```

5. Compile your package.

Execute in Shell #1

In [ ]:

```
cd ~/ros2_ws
```

In [ ]:

```
colcon build --packages-select publisher_pkg
```

In [ ]:

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

6. Finally, launch the publisher node in your shell.

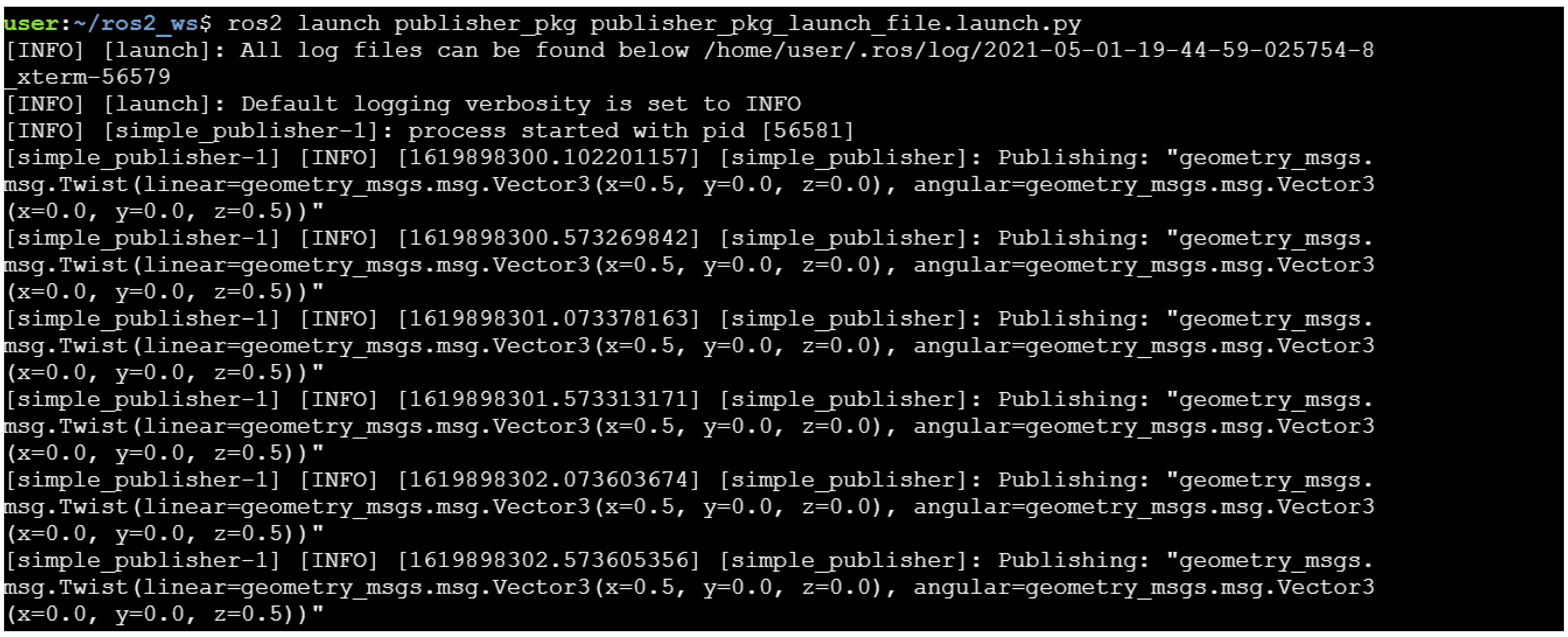
Execute in Shell #1

In [ ]:

```
ros2 launch publisher_pkg publisher_pkg_launch_file.launch.py
```

You should get a result similar to the following:

Shell #1 Output



```
user:~/ros2_ws$ ros2 launch publisher_pkg publisher_pkg_launch_file.launch.py
[INFO] [launch]: All log files can be found below /home/user/.ros/log/2021-05-01-19-44-59-025754-8
_xterm-56579
[INFO] [launch]: Default logging verbosity is set to INFO
[INFO] [simple_publisher-1]: process started with pid [56581]
[simple_publisher-1] [INFO] [1619898300.102201157] [simple_publisher]: Publishing: "geometry_msgs.
msg.Twist(linear=geometry_msgs.msg.Vector3(x=0.5, y=0.0, z=0.0), angular=geometry_msgs.msg.Vector3
(x=0.0, y=0.0, z=0.5))"
[simple_publisher-1] [INFO] [1619898300.573269842] [simple_publisher]: Publishing: "geometry_msgs.
msg.Twist(linear=geometry_msgs.msg.Vector3(x=0.5, y=0.0, z=0.0), angular=geometry_msgs.msg.Vector3
(x=0.0, y=0.0, z=0.5))"
[simple_publisher-1] [INFO] [1619898301.073378163] [simple_publisher]: Publishing: "geometry_msgs.
msg.Twist(linear=geometry_msgs.msg.Vector3(x=0.5, y=0.0, z=0.0), angular=geometry_msgs.msg.Vector3
(x=0.0, y=0.0, z=0.5))"
[simple_publisher-1] [INFO] [1619898301.573313171] [simple_publisher]: Publishing: "geometry_msgs.
msg.Twist(linear=geometry_msgs.msg.Vector3(x=0.5, y=0.0, z=0.0), angular=geometry_msgs.msg.Vector3
(x=0.0, y=0.0, z=0.5))"
[simple_publisher-1] [INFO] [1619898302.073603674] [simple_publisher]: Publishing: "geometry_msgs.
msg.Twist(linear=geometry_msgs.msg.Vector3(x=0.5, y=0.0, z=0.0), angular=geometry_msgs.msg.Vector3
(x=0.0, y=0.0, z=0.5))"
[simple_publisher-1] [INFO] [1619898302.573605356] [simple_publisher]: Publishing: "geometry_msgs.
msg.Twist(linear=geometry_msgs.msg.Vector3(x=0.5, y=0.0, z=0.0), angular=geometry_msgs.msg.Vector3
(x=0.0, y=0.0, z=0.5))"
```

As you learned in the previous section, to see the current status of the Topic where you are posting the message, execute the command `ros2 topic echo`.

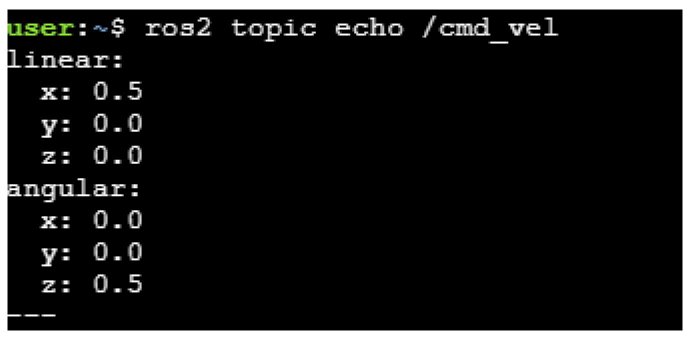
Execute in Shell #2

In [ ]:

```
ros2 topic echo /cmd_vel
```

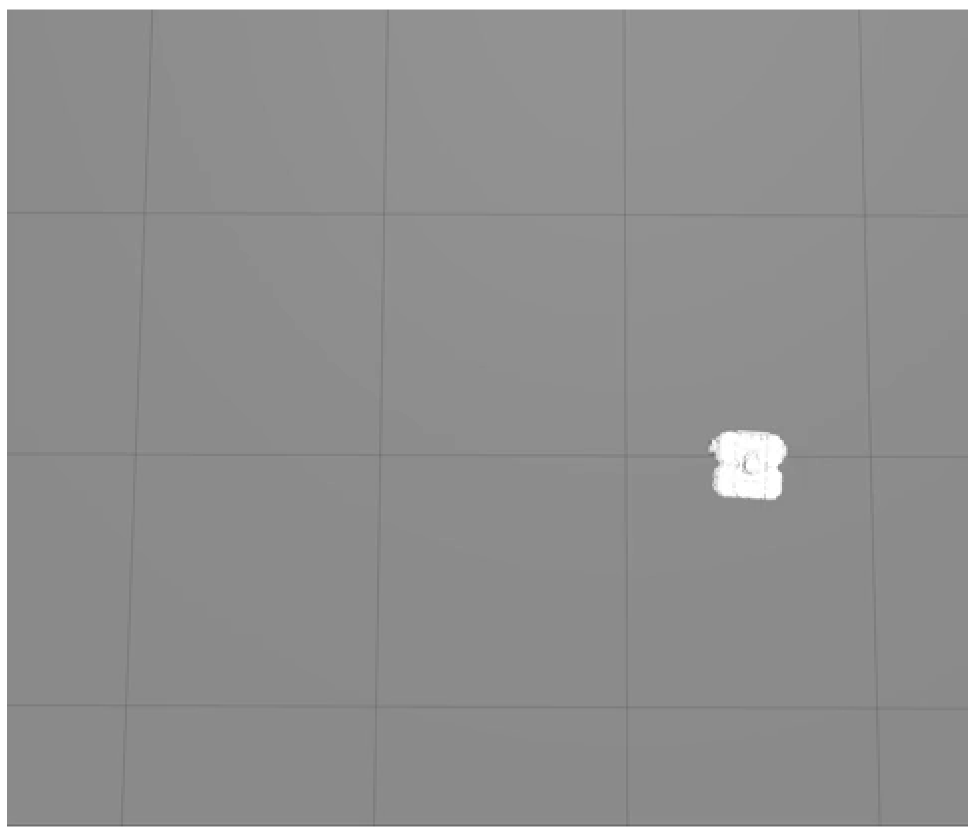
You should see output similar to the following in the terminal:

Shell #2 Output



```
user:~$ ros2 topic echo /cmd_vel
linear:
  x: 0.5
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 0.5
---
```

In this case, you continuously post messages in the cmd_vel Topic to print movement on the simulation robot. For example, in the Gazebo simulator, you should get a result similar to the following:



For now, terminate the command using `Ctrl+C` in `Shell #1` and stop the robot's movement using a `ros2 topic pub` command in the same shell.

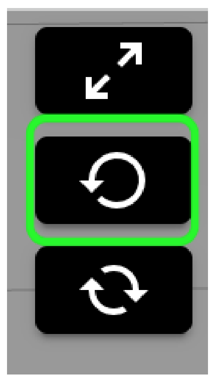
Execute in Shell #1

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

- End of Example 3.4 -

- Notes -

For testing purposes, remember that you can send the robot back to its initial position by clicking on the `Reset the simulation` button:



Or you can also do it directly through the command line with the following command:

In [ ]:
```
ros2 service call /reset_world std_srvs/srv/Empty {}
```

- End of Notes -

3.2.2 Code Review

As you should have noticed, all the code is explained in the code comments. However, have an extra look at the most important parts of the code you have just executed.

Initiate a node named **simple_publisher**:

In [ ]:
```
super().__init__('simple_publisher')
```

Create a Publisher for the `/cmd_vel` Topic that uses `Twist` messages:

In [ ]:
```
self.publisher_ = self.create_publisher(Twist, 'cmd_vel', 10)
```

Create a timer object that will trigger the `timer_callback` method each 0.5 (`timer_period`) seconds:

In [ ]:
```
self.timer = self.create_timer(timer_period, self.timer_callback)
```

Publish the contents of `msg` into the `/cmd_vel` Topic.

In [ ]:
```
self.publisher_.publish(msg)
```

3.3 Topic Subscriber

In the previous section, you worked with Publishers. Through that, you successfully moved your simulated robot in the Gazebo simulation, but it doesn't make much sense to move it just like that. So instead, you are probably thinking about moving it in different directions depending on the position in which it is or the objects around it. For that, you will need to use **Subscribers** to get some additional data.

To get started, you should be clear about what a subscriber is. Considering everything you have seen so far, I propose that you see it as a node that reads information from a given Topic.

3.3.1 Create a Simple Subscriber Node

- Example 3.5 -

1. Create a new package named **subscriber_pkg**. Follow the steps from the previous section to create a publisher node package.

Execute in Shell #1

In [ ]:
```
ros2 pkg create --build-type ament_python subscriber_pkg --dependencies rclpy std_msgs sensor_msgs
```

In this case, you will also add the `std_msgs` and `sensor_msgs` dependencies.

- Notes -

You're likely asking yourself, **`"How do I know which interface to add for each topic I want to use?"`**

In response to your question, I propose that you use what you learned in the first section of this unit, namely `ros2 topic info <topic_name>`. You then know which interface works with which node, and, consequently, which one you need to include when creating a new package. Even if you want to know more, you could try the `ros2 interface` commands that you saw earlier in this unit.

- End of Notes -

2. Create a new file named **simple_subscriber.py** inside the **subscriber_pkg** folder in the package you have just created.

Inside the file, copy the following code:

simple_subscriber.py

In [ ]:

```python
import rclpy
# import the ROS2 python libraries
from rclpy.node import Node
# import the LaserScan module from sensor_msgs interface
from sensor_msgs.msg import LaserScan
# import Quality of Service library, to set the correct profile and reliability to read sensor data.
from rclpy.qos import ReliabilityPolicy, QoSProfile


class SimpleSubscriber(Node):

    def __init__(self):
        # Here you have the class constructor
        # call super() in the constructor to initialize the Node object
        # the parameter you pass is the node name
        super().__init__('simple_subscriber')
        # create the subscriber object
        # in this case, the subscriptor will be subscribed on /scan topic with a queue size of 10 messages.
        # use the LaserScan module for /scan topic
        # send the received info to the listener_callback method.
        self.subscriber = self.create_subscription(
            LaserScan,
            '/scan',
            self.listener_callback,
            QoSProfile(depth=10, reliability=ReliabilityPolicy.RELIABLE))  # is the most used to read LaserScan data and some sensor data.

    def listener_callback(self, msg):
        # print the log info in the terminal
        self.get_logger().info('I receive: "%s"' % str(msg))


def main(args=None):
    # initialize the ROS communication
    rclpy.init(args=args)
    # declare the node constructor
    simple_subscriber = SimpleSubscriber()
    # pause the program execution, waits for a request to kill the node (ctrl+c)
    rclpy.spin(simple_subscriber)
    # Explicity destroy the node
    simple_subscriber.destroy_node()
    # shutdown the ROS communication
    rclpy.shutdown()


if __name__ == '__main__':
    main()
```

3. Create a launch file named **subscriber_pkg_launch_file.launch.py** to launch the node you have just created.

To do so, execute the following codes as you have done before:

Execute in Shell #1

In [ ]:

```
cd ~/ros2_ws/src/subscriber_pkg
```

In [ ]:

```
mkdir launch
```

Execute in Shell #1

In [ ]:

```
cd ~/ros2_ws/src/subscriber_pkg/launch
```

In [ ]:

```
touch subscriber_pkg_launch_file.launch.py
```

In [ ]:

```
chmod +x subscriber_pkg_launch_file.launch.py
```

Inside the newly created launch file, write the necessary code to launch the executable files of the **simple_subscriber** script.

subscriber_pkg_launch_file.launch.py

In [ ]:

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

def generate_launch_description():
    return LaunchDescription([
        Node(
            package='subscriber_pkg',
            executable='simple_subscriber',
            output='screen'),
    ])
```

4. Modify the setup.py to install the launch file that you have just created, and add the entry points to the executable for the **simple_subscriber.py** script.

setup.py

In [ ]:

```
from setuptools import setup
import os
from glob import glob

package_name = 'subscriber_pkg'

setup(
    name=package_name,
    version='0.0.0',
    packages=[package_name],
    data_files=[
        ('share/ament_index/resource_index/packages',
            ['resource/' + package_name]),
        ('share/' + package_name, ['package.xml']),
        (os.path.join('share', package_name), glob('launch/*.launch.py'))
    ],
    install_requires=['setuptools'],
    zip_safe=True,
    maintainer='somebody very awesome',
    maintainer_email='user@user.com',
    description='TODO: Package description',
    license='TODO: License declaration',
    tests_require=['pytest'],
    entry_points={
        'console_scripts': [
            'simple_subscriber = subscriber_pkg.simple_subscriber:main'
        ],
    },
)
```

5. Compile your package.

Execute in Shell #1

In [ ]:

```
cd ~/ros2_ws
```

In [ ]:

```
colcon build --packages-select subscriber_pkg
```

In [ ]:

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

7.- Finally, launch the subscriber node in your shell.

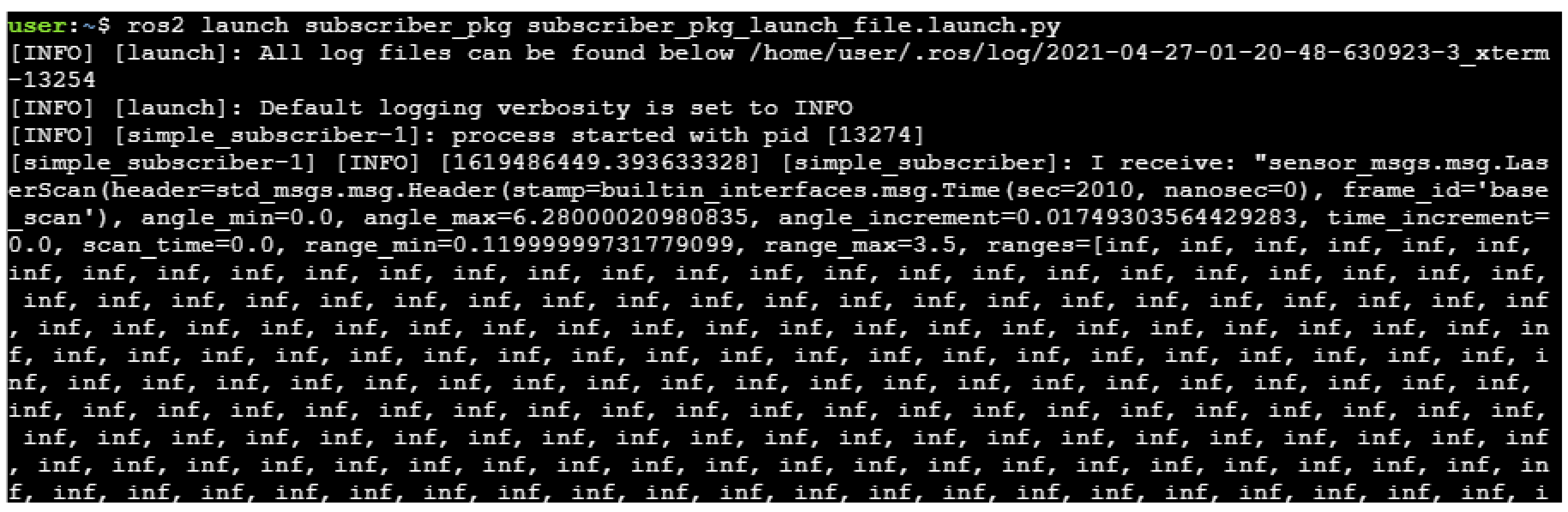
Execute in Shell #1

In [ ]:

```
ros2 launch subscriber_pkg subscriber_pkg_launch_file.launch.py
```

You should see the output in your shell that is similar to the following:

Shell #1 Output



```
user:~$ ros2 launch subscriber_pkg subscriber_pkg_launch_file.launch.py
[INFO] [launch]: All log files can be found below /home/user/.ros/log/2021-04-27-01-20-48-630923-3_xterm
-13254
[INFO] [launch]: Default logging verbosity is set to INFO
[INFO] [simple_subscriber-1]: process started with pid [13274]
[simple_subscriber-1] [INFO] [1619486449.393633328] [simple_subscriber]: I receive: "sensor_msgs.msg.Las
erScan(header=std_msgs.msg.Header(stamp=builtin_interfaces.msg.Time(sec=2010, nanosec=0), frame_id='base
_scan'), angle_min=0.0, angle_max=6.28000020980835, angle_increment=0.01749303564429283, time_increment=
0.0, scan_time=0.0, range_min=0.11999999731779099, range_max=3.5, ranges=[inf, inf, inf, inf, inf, inf,
inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf,
 inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf
, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, in
f, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, i
nf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf,
inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf,
 inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf
, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, in
f, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, i
```

- End of Example 3.5 -

3.3.2 Code Review

As you should have noticed, all the code is explained in the code comments. However, have an extra look at the most important parts of the code you have just executed.

Create a Subscriber for the `/scan` topic that uses `LaserScan` messages:

In [ ]:

```
self.subscriber= self.create_subscription(
            LaserScan,
            '/scan',
            self.listener_callback,
            QoSProfile(depth=10, reliability=ReliabilityPolicy.RELIABLE))
```

Notice that you are doing two interesting things here:

- First, you define a callback method named `listener_callback` for the subscriber. This means that every time a message is published to the `/scan` topic, this method will be triggered.
- Second, you define a `QoSProfile`. This refers to the `Quality of Service` of the topic.

- Notes -

Quality of Service. You may have read this somewhere, but where? You might have found some related things if you searched and played more with the `ros2 topic echo` command.

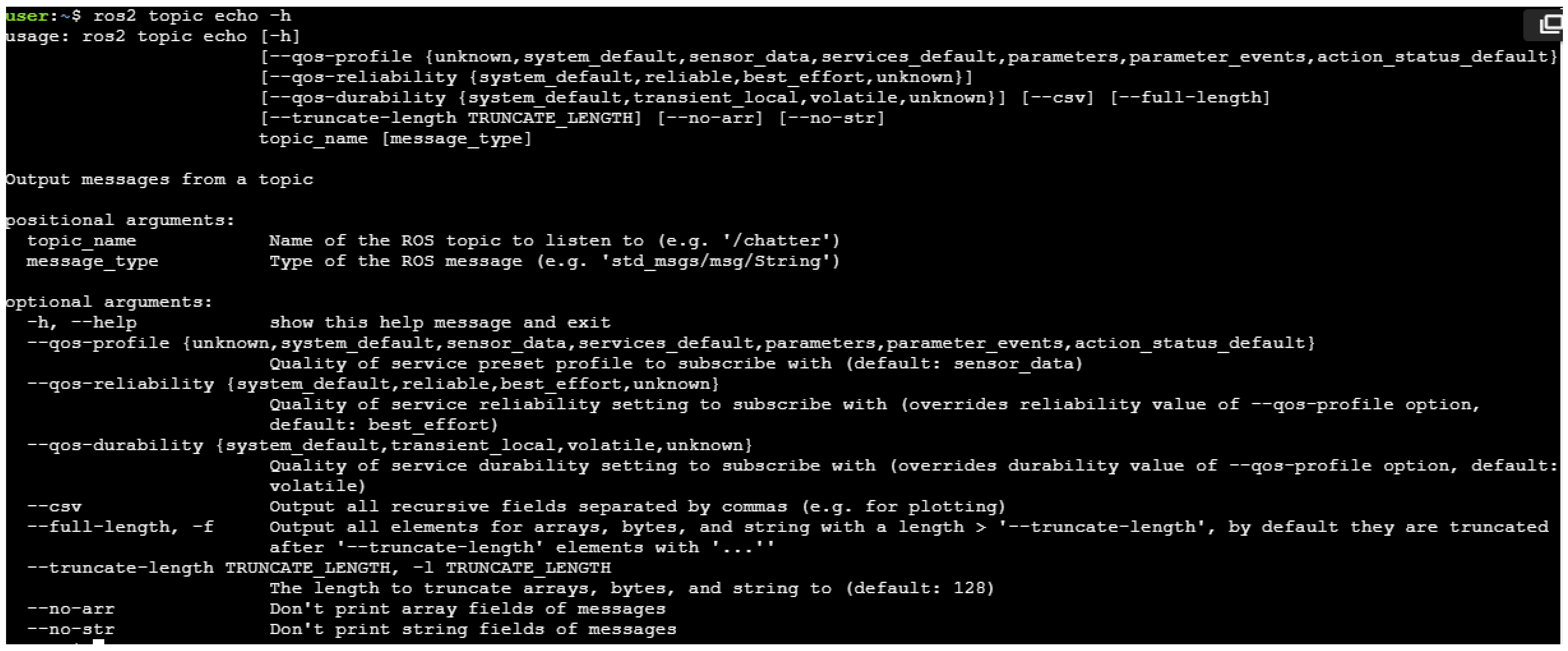
Execute in Shell #1

In [ ]:

```
ros2 topic echo -h
```

All these options, including Quality of Service, are referred to as **QoS**.

Shell #1 Output



```
user:~$ ros2 topic echo -h
usage: ros2 topic echo [-h]
                       [--qos-profile {unknown,system_default,sensor_data,services_default,parameters,parameter_events,action_status_default}]
                       [--qos-reliability {system_default,reliable,best_effort,unknown}]
                       [--qos-durability {system_default,transient_local,volatile,unknown}] [--csv] [--full-length]
                       [--truncate-length TRUNCATE_LENGTH] [--no-arr] [--no-str]
                       topic_name [message_type]

Output messages from a topic

positional arguments:
  topic_name            Name of the ROS topic to listen to (e.g. '/chatter')
  message_type          Type of the ROS message (e.g. 'std_msgs/msg/String')

optional arguments:
  -h, --help            show this help message and exit
  --qos-profile {unknown,system_default,sensor_data,services_default,parameters,parameter_events,action_status_default}
                        Quality of service preset profile to subscribe with (default: sensor_data)
  --qos-reliability {system_default,reliable,best_effort,unknown}
                        Quality of service reliability setting to subscribe with (overrides reliability value of --qos-profile option,
                        default: best_effort)
  --qos-durability {system_default,transient_local,volatile,unknown}
                        Quality of service durability setting to subscribe with (overrides durability value of --qos-profile option, default:
                        volatile)
  --csv                 Output all recursive fields separated by commas (e.g. for plotting)
  --full-length, -f     Output all elements for arrays, bytes, and string with a length > '--truncate-length', by default they are truncated
                        after '--truncate-length' elements with '...''
  --truncate-length TRUNCATE_LENGTH, -l TRUNCATE_LENGTH
                        The length to truncate arrays, bytes, and string to (default: 128)
  --no-arr              Don't print array fields of messages
  --no-str              Don't print string fields of messages
```

As you can see, you can select different profiles such as `sensor_data`, `services_default`, `parameters`, etc.

For now, all you need to know is that the QoS settings of the topic Publisher and the Subscriber node need to be compatible. If they are not compatible, the communication between them won't work.

To get data about the QoS setting of a specific topic, you can use the following command:

Execute in Shell #1

In [ ]:

```
ros2 topic info /scan -v
```

Shell #1 Output

```
Type: sensor_msgs/msg/LaserScan

Publisher count: 1

Node name: turtlebot3_laserscan
Node namespace: /
Topic type: sensor_msgs/msg/LaserScan
Endpoint type: PUBLISHER
GID: 01.0f.8e.74.10.01.8f.72.01.00.00.00.00.00.3e.03.00.00.00.00.00.00.00.00
QoS profile:
  Reliability: RELIABLE
  History (Depth): UNKNOWN
  Durability: VOLATILE
  Lifespan: Infinite
  Deadline: Infinite
  Liveliness: AUTOMATIC
  Liveliness lease duration: Infinite

Subscription count: 0
```

For instance, you can see from the output that the **`Reliability`** setting is **`RELIABLE`**. So this is why you are setting the same in your Subscriber node:

In [ ]:

```
QoSProfile(depth=10, reliability=ReliabilityPolicy.RELIABLE))
```

In any case, do not worry too much about these QoS settings right now, since they are a concept a little bit more advanced. If you want to learn more about this topic, you can have a look at the **Understanding QoS** unit from the Intermediate ROS2 course.

- End of Notes -

Inside the **`listener_callback`** method, all you do is print the received `LaserScan` message:

In [ ]:

```
self.get_logger().info('I receive: "%s"' % str(msg))
```

3.4 Write a Publisher & Subscriber Node

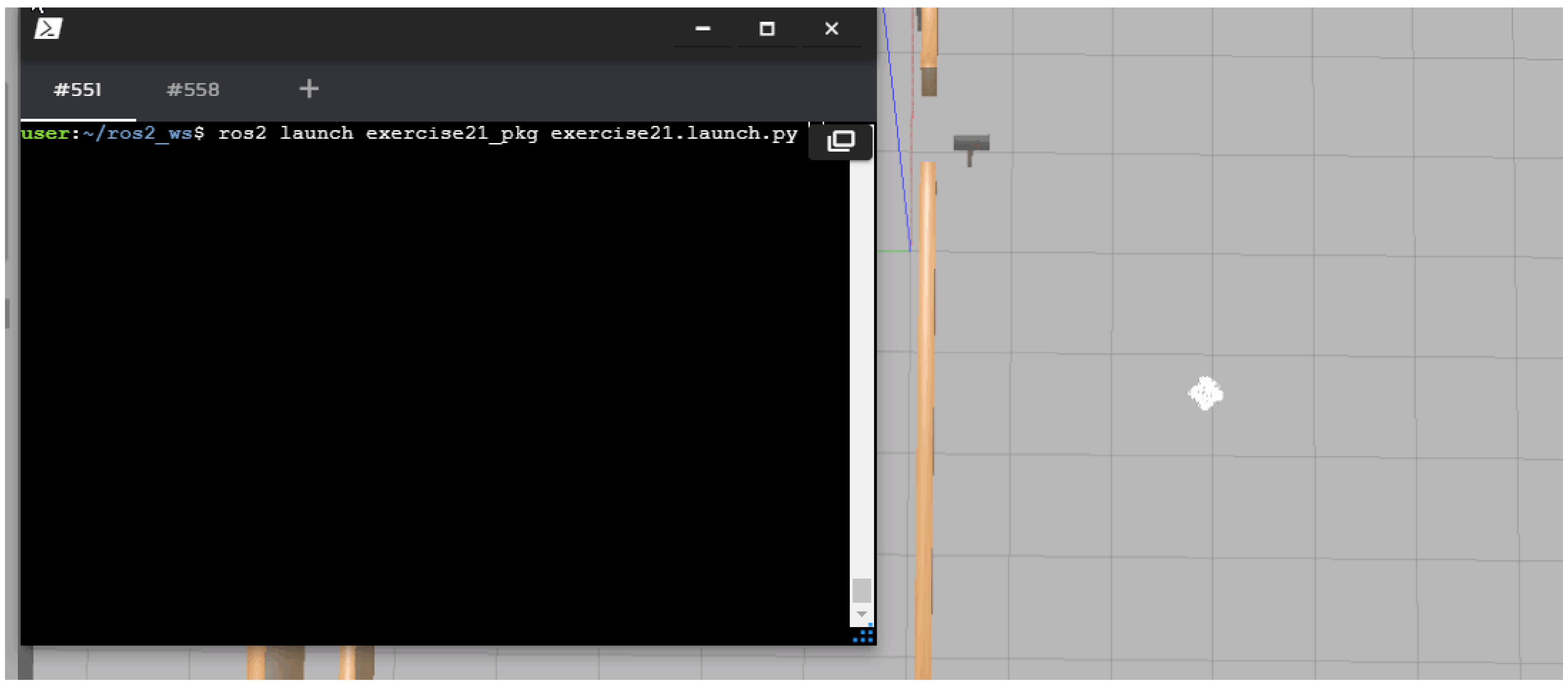
Now that you know how to work with Subscribers and Publishers, why not try combining them to accomplish a specific task? It is up to you now.

- Exercise 3.1 -

Write a Publisher and Subscriber node:

- Create a new package and call it **exercise31_pkg** with `rclpy`, `std_msgs`, `sensor_msgs`, and `geometry_msgs` as dependencies.
- Use the previous publisher code as a template to create a new program that makes the robot turn left using **`Twist`** messages.
- Get information from the laser and use the **`LaserScan`** messages to decide how to move.
  - Turn left until you get a value lower than 5m (from the laser direction in front of the robot). Then, move straight.
  - If the distance received is greater than 50cm, move in a straight line towards the wall (do not go too fast!).
  - If the distance received is less than 50cm, the robot stops.

The resulting motion should be something similar to this:



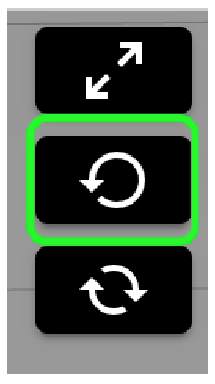
```
user:~/ros2_ws$ ros2 launch exercise21_pkg exercise21.launch.py
```

- Create the launch file to start your program.

- End of Exercise 3.1 -

- Notes -

For testing purposes, remember that you can send the robot back to its initial position by clicking on the `Reset the simulation` button:



Or you can also do it directly through the command line with the following command:

In [ ]:

```
ros2 service call /reset_world std_srvs/srv/Empty {}
```

- End of Notes -

You can compare the results you got with the solution below.

- Solution for Exercise 3.1 -

1. Create a new package named **exercise31_pkg**.

In [ ]:

```
ros2 pkg create --build-type ament_python exercise31_pkg --dependencies rclpy std_msgs sensor_msgs geometry_msgs
```

2. Create a new file named **exercise31.py** inside the **exercise31_pkg** folder in the package you have just created.

exercise31.py

In [ ]:

```
import rclpy
# import the ROS2 python libraries
from rclpy.node import Node
# import the Twist module from geometry_msgs interface
from geometry_msgs.msg import Twist
# import the LaserScan module from sensor_msgs interface
from sensor_msgs.msg import LaserScan
from rclpy.qos import ReliabilityPolicy, QoSProfile

class Exercise31(Node):

    def __init__(self):
        # Here you have the class constructor
        # call the class constructor
        super().__init__('exercise31')
        # create the publisher object
        self.publisher_ = self.create_publisher(Twist, 'cmd_vel', 10)
        # create the subscriber object
        self.subscriber = self.create_subscription(LaserScan, '/scan', self.laser_callback, QoSProfile(depth=10, reliability=ReliabilityPolicy.RELIABLE))
        # define the timer period for 0.5 seconds
        self.timer_period = 0.5
        # define the variable to save the received info
        self.laser_forward = 0
        # create a Twist message
        self.cmd = Twist()
        self.timer = self.create_timer(self.timer_period, self.motion)

    def laser_callback(self,msg):
        # Save the frontal laser scan info at 0°
        self.laser_forward = msg.ranges[359]


    def motion(self):
        # print the data
        self.get_logger().info('I receive: "%s"' % str(self.laser_forward))
        # Logic of move
        if self.laser_forward > 5:
            self.cmd.linear.x = 0.5
            self.cmd.angular.z = 0.5
        elif self.laser_forward < 5 and self.laser_forward >= 0.5:
            self.cmd.linear.x = 0.2
            self.cmd.angular.z = 0.0
        else:
            self.cmd.linear.x = 0.0
            self.cmd.angular.z = 0.0

        # Publishing the cmd_vel values to a Topic
        self.publisher_.publish(self.cmd)



def main(args=None):
    # initialize the ROS communication
    rclpy.init(args=args)
    # declare the node constructor
    exercise31 = Exercise31()
    # pause the program execution, waits for a request to kill the node (ctrl+c)
    rclpy.spin(exercise31)
    # Explicity destroy the node
    exercise31.destroy_node()
    # shutdown the ROS communication
    rclpy.shutdown()

if __name__ == '__main__':
    main()
```

3. Create a launch file named **exercise31_pkg_launch_file.launch.py** to launch the node you have just created.

Execute in Shell #1

In [ ]:

```
cd ~/ros2_ws/src/exercise31_pkg
```

In [ ]:

```
mkdir launch
```

Execute in Shell #1

In [ ]:

```
cd ~/ros2_ws/src/exercise31_pkg/launch
```

In [ ]:

```
touch exercise31_pkg_launch_file.launch.py
```

In [ ]:

```
chmod +x exercise31_pkg_launch_file.launch.py
```

Add the following code to the file **exercise31_pkg_launch_file.launch.py** you have just created.

exercise31_pkg_launch_file.launch.py

In [ ]:

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

def generate_launch_description():
    return LaunchDescription([
        Node(
            package='exercise31_pkg',
            executable='exercise31',
            output='screen'),
    ])
```

4. Modify the setup.py to add the launch file, which you have just created, and the entry points to the executable for the **exercise31** script.

setup.py

In [ ]:

```
from setuptools import setup
import os
from glob import glob

package_name = 'exercise31_pkg'

setup(
    name=package_name,
    version='0.0.0',
    packages=[package_name],
    data_files=[
        ('share/ament_index/resource_index/packages',
            ['resource/' + package_name]),
        ('share/' + package_name, ['package.xml']),
        (os.path.join('share', package_name), glob('launch/*.launch.py'))
    ],
    install_requires=['setuptools'],
    zip_safe=True,
    maintainer='somebody very awesome',
    maintainer_email='user@user.com',
    description='TODO: Package description',
    license='TODO: License declaration',
    tests_require=['pytest'],
    entry_points={
        'console_scripts': [
            'exercise31 = exercise31_pkg.exercise31:main'
        ],
    },
)
```

5. Compile your package.

Execute in Shell #1

In [ ]:

```
cd ~/ros2_ws
```

In [ ]:

```
colcon build --packages-select exercise31_pkg
```

In [ ]:

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

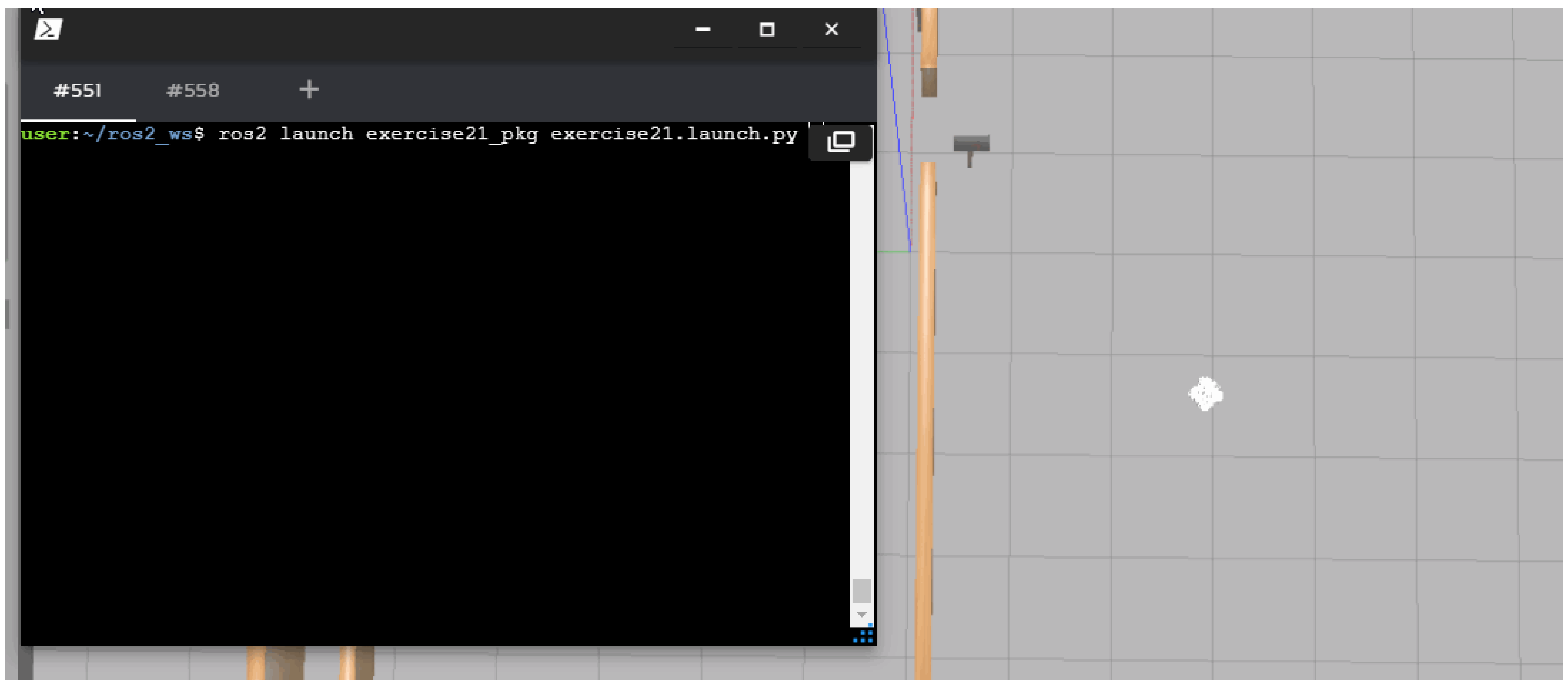
6. Finally, launch the publisher node in your shell.

Execute in Shell #1

In [ ]:

```
ros2 launch exercise31_pkg exercise31_pkg_launch_file.launch.py
```

You should receive a result similar to the image below:



```
user:~/ros2_ws$ ros2 launch exercise21_pkg exercise21.launch.py
```

- End of solution for Exercise 3.1 -

3.4.1 Code Review

As you should have noticed, all the code is explained in the code comments. First, however, have a second look at the most important parts of the code you have just executed.

This time, create both a Publisher and a Subscriber in your node (because you will move the robot depending on the data you receive from the laser):

In [ ]:

```
self.publisher_ = self.create_publisher(Twist, 'cmd_vel', 10)

self.subscriber = self.create_subscription(LaserScan, '/scan', self.laser_callback, QoSProfile(depth=10, reliability=ReliabilityPolicy.RELIABLE))
```

You also create a timer object with a method named `motion`, which you will use to program the logic of your node:

In [ ]:

```
self.timer = self.create_timer(self.timer_period, self.motion)
```

You get the data of the laser corresponding to the front of the robot and store it in `self.laser_forward`:

In [ ]:

```
self.laser_forward = msg.ranges[359]
```

The logic of our node:

In [ ]:

```
if self.laser_forward > 5:
    self.cmd.linear.x = 0.5
    self.cmd.angular.z = 0.5
elif self.laser_forward < 5 and self.laser_forward >= 0.5:
    self.cmd.linear.x = 0.2
    self.cmd.angular.z = 0.0
else:
    self.cmd.linear.x = 0.0
    self.cmd.angular.z = 0.0

# Publishing the cmd_vel values to the Topic
self.publisher_.publish(self.cmd)
```

- If the distance received from the laser is higher than 5, turn to the left.
- Move forward if the distance received from the laser is lower than 5 but higher than 0.5.
- If none of the previous conditions are met, stop the robot.

3.5 How to Create a Custom Interface

It is always recommended to use the interfaces that ROS2 already provides. Remember you can check all ROS2 available interfaces using the `ros2 interface list`) command. However, if none fits your needs, you can create a new one.

Create a new package named **custom_interfaces**. This package, however, has to be a **CMake package**. Currently, there is no way to generate custom interfaces in a pure Python package. However, you can create a custom interface in a CMake package and then use it in a Python node.

Execute in Shell #1

In [ ]:

```
cd ~/ros2_ws/src
```

In [ ]:

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

- Notes -

Now, specify this is a CMake package with the `--build-type` flag set to `ament_cmake`.

- End of Notes -

Once you have created this new package, you will create a new message.

To create a new interface, you have to follow the next 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 the `CMakeLists.txt` file (more information below)
4. Modify `package.xml` file (more information below)
5. Compile and source
6. Use in your node

For practice, create a new interface that indicates age with years, months, and days.

1. Create a directory `msg` in your package.

In [ ]:

```
cd ~/ros2_ws/src/custom_interfaces
```

In [ ]:

```
mkdir msg
```

2. The Age.msg file must contain the following:

Age.msg

In [ ]:

```
int32 year
int32 month
int32 day
```

3. Modify the CMakeLists.txt file:

You have to edit two functions inside `CMakeLists.txt`:

find_package()

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

In [ ]:

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

rosidl_generate_interfaces()

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

In [ ]:

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

In summarizing, this is the minimum expression of what is needed for the `CMakelists.txt` to generate a new interface:

In [ ]:

```
cmake_minimum_required(VERSION 3.8)
project(custom_interfaces)

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

# find dependencies
find_package(ament_cmake REQUIRED)
find_package(rclcpp REQUIRED)
find_package(std_msgs REQUIRED)
find_package(rosidl_default_generators REQUIRED)
# uncomment the following section in order to fill in
# further dependencies manually.
# find_package(<dependency> REQUIRED)

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

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

ament_package()
```

3. Modify the package.xml file:

Add the following lines to the `package.xml` file:

In [ ]:

```
<build_depend>rosidl_default_generators</build_depend>

<exec_depend>rosidl_default_runtime</exec_depend>

<member_of_group>rosidl_interface_packages</member_of_group>
```

This is the minimum expression of the `package.xml`.

In [ ]:

```
<?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>custom_interfaces</name>
  <version>0.0.0</version>
  <description>TODO: Package description</description>
  <maintainer email="ubuntu@todo.todo">ubuntu</maintainer>
  <license>TODO: License declaration</license>

  <buildtool_depend>ament_cmake</buildtool_depend>

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

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

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

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

1. Now, compile the package to generate the new interface.

Execute in Shell #1

In [ ]:

```
cd ~/ros2_ws
```

In [ ]:

```
colcon build --packages-select custom_interfaces
```

In [ ]:

```
source install/setup.bash
```

- VERY IMPORTANT -

VERY IMPORTANT: When you compile new messages, there is an extra step before using 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 build process.

If you do not do this, it might give you an import error, saying it does not find the message generated.

- END -

To verify that your message has been created successfully, type into your Webshell: `ros2 interface show custom_interfaces/msg/Age`.

If the structure of the age message appears, it means that your message has been created successfully and is ready to be used in your ROS2 programs.

Execute in Shell #1

In [ ]:

```
ros2 interface show custom_interfaces/msg/Age
```

Shell #1 Output

In [ ]:

```
int32 year
int32 month
int32 day
```

3.6 Use a Custom Interface

- Example 3.6 -

1. Create a new package named **example36_pkg** using python.

In [ ]:

```
ros2 pkg create --build-type ament_python example36_pkg --dependencies rclpy std_msgs geometry_msgs custom_interfaces
```

Did you see it? Note that you have added the package **custom_interface** as a dependency of your new package.

- Notes -

If you forget to add the dependency in the creation process, you can add a package dependency manually in the following way:

Because you are using a Python package, you do not have a **CMakeLists.txt**. You only have to add the following in the **package.xml**.

In [ ]:

```
<depend>custom_interfaces</depend>
```

- End of Notes -

Now, create a Python program based on Exercise 3.1, but this time you will use your new interface to include the date this program was made.

Go to your package. Inside the **example36_pkg** folder, create a new script called **example36.py**. Copy the following code:

example36.py

In [ ]:

```python
import rclpy
from rclpy.node import Node
from rclpy.qos import ReliabilityPolicy, QoSProfile
# Import custom interface Age
from custom_interfaces.msg import Age


class Example36(Node):

    def __init__(self):
        # Here you have the class constructor
        # call the class constructor
        super().__init__('example36')
        # create the publisher object
        self.publisher_ = self.create_publisher(Age, 'age', 10)
        # create an Age message
        self.age = Age()
        # define the timer period for 0.5 seconds
        self.timer_period = 0.5
        self.timer = self.create_timer(self.timer_period, self.timer_callback)

    def timer_callback(self):
        # create an Age message
        self.age.year = 2031
        self.age.month = 5
        self.age.day = 21
        # publish the Age message
        self.publisher_.publish(self.age)


def main(args=None):
    # initialize the ROS communication
    rclpy.init(args=args)
    # declare the node constructor
    example36 = Example36()
    rclpy.spin(example36)
    # Explicity destroy the node
    example36.destroy_node()
    # shutdown the ROS communication
    rclpy.shutdown()


if __name__ == '__main__':
    main()
```

Now, have a closer look at the new additions to the code.

First, you can see that you have imported your new interface.

In [ ]:

```python
from custom_interfaces.msg import Age
```

Also, you have to create the Age msg:

In [ ]:

```python
self.age = Age()
```

Finally, you have to use it.

In [ ]:

```python
def timer_callback(self):
    # create an Age message
    self.age.year = 2031
    self.age.month = 5
    self.age.day = 21
    # publish the Age message
    self.publisher_.publish(self.age)
```

You use every part of your interface to create a date and publish it into the `/age` topic.

3. Create a launch file named **example36.launch.py** to launch the service server node you just created.

Execute in Shell #1

In [ ]:

```
cd ~/ros2_ws/src/example36_pkg
```

In [ ]:

```
mkdir launch
```

In [ ]:

```
cd launch
```

In [ ]:

```
touch example36.launch.py
```

In [ ]:

```
chmod +x example36.launch.py
```

Inside **example36.launch.py**, write the necessary code to launch the executable files of the **example36** script.

example36.launch.py

In [ ]:

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

def generate_launch_description():
    return LaunchDescription([
        Node(
            package='example36_pkg',
            executable='example36',
            output='screen'),
    ])
```

4. Modify the setup.py to add the launch file you have just created and the entry points to the executables scripts.

setup.py

In [ ]:

```
from setuptools import setup
import os
from glob import glob

package_name = 'example36_pkg'

setup(
    name=package_name,
    version='0.0.0',
    packages=[package_name],
    data_files=[
        ('share/ament_index/resource_index/packages',
            ['resource/' + package_name]),
        ('share/' + package_name, ['package.xml']),
        (os.path.join('share', package_name), glob('launch/*.launch.py'))
    ],
    install_requires=['setuptools'],
    zip_safe=True,
    maintainer='user',
    maintainer_email='user@todo.todo',
    description='TODO: Package description',
    license='TODO: License declaration',
    tests_require=['pytest'],
    entry_points={
        'console_scripts': [
        'example36 = example36_pkg.example36:main'
        ],
    },
)
```

5. Compile your package.

Execute in Shell #1

In [ ]:

```
cd ~/ros2_ws
```

In [ ]:

```
colcon build --packages-select example36_pkg
```

In [ ]:

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

6. Launch the service node in your shell.

Execute in Shell #1

In [ ]:

```
ros2 launch example36_pkg example36.launch.py
```

Execute in Shell #2

In [ ]:

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

In [ ]:

```
ros2 topic echo /age
```

Whenever you stop your program with `Ctrl+C`, you should get a result similar to the following:

Shell #2 Output

```
year: 2031
month: 5
day: 21
---
year: 2031
month: 5
day: 21
---
year: 2031
month: 5
day: 21
---
```

And that's all! You can now create and use your own interfaces. **Congratulations!**

- End of Example 3.5 -

3.7 Topics Quiz

You will take a small quiz to put everything together with all you have learned during this course. Subscribers, Publisher, Messages - you will need to use these concepts to succeed!

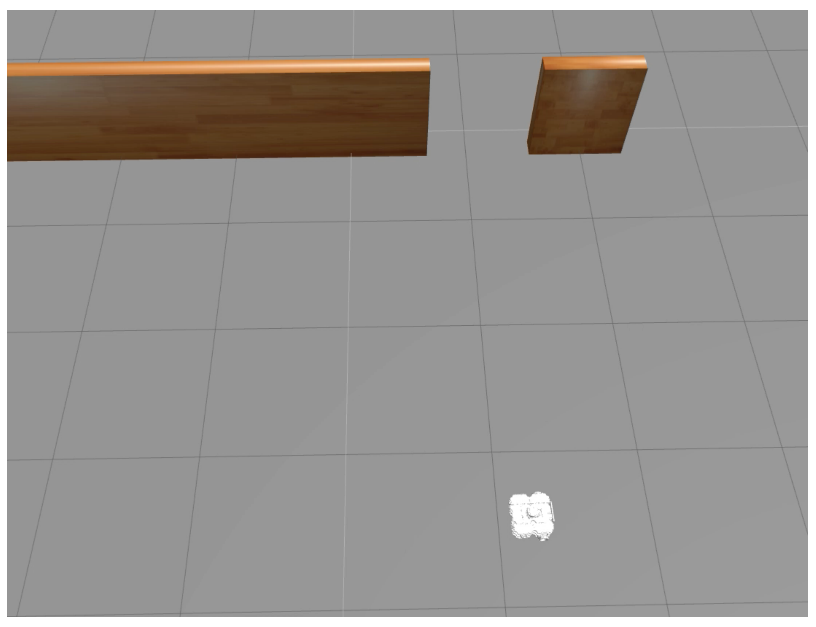
To evaluate this quiz, you will perform different tasks. For each task, **precise instructions** are provided: name of the package, names of the launch files and Python scripts, topic names to use, etc.

It is **VERY IMPORTANT** that you strictly follow these instructions since they will allow our automated correction system to score your quiz properly. If your name differs from the one specified in the exam instructions, your exercise will be marked as **FAILED**, even though it works correctly.

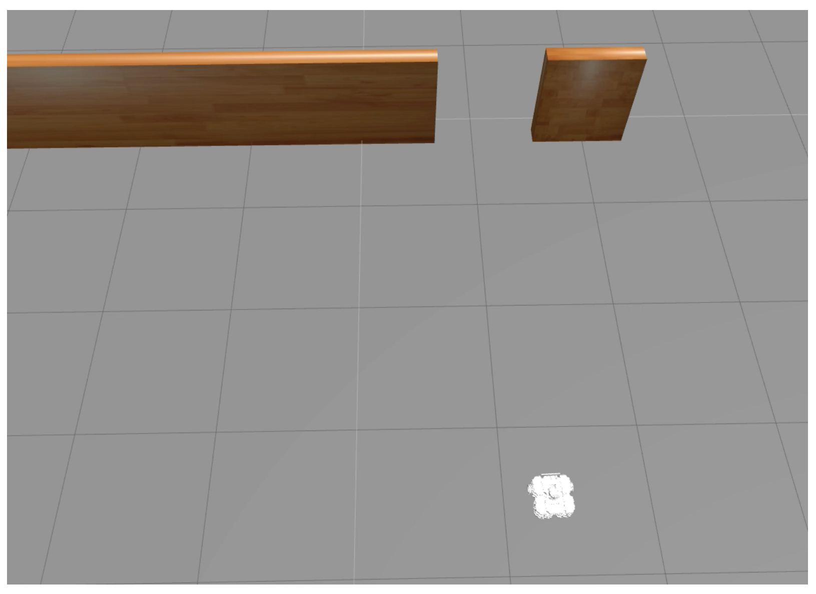
1. Create a Publisher that writes into the `/cmd_vel` topic to move the robot.
2. Create a Subscriber that reads from the `/odom` topic. This is the topic where the odometry publishes its data.
3. Depending on the readings you receive from the odometry topic, you will have to change the data you send to the `/cmd_vel` topic to follow a specific trajectory. This means, using the values of the odometry to decide.

Your program should follow the following logic:

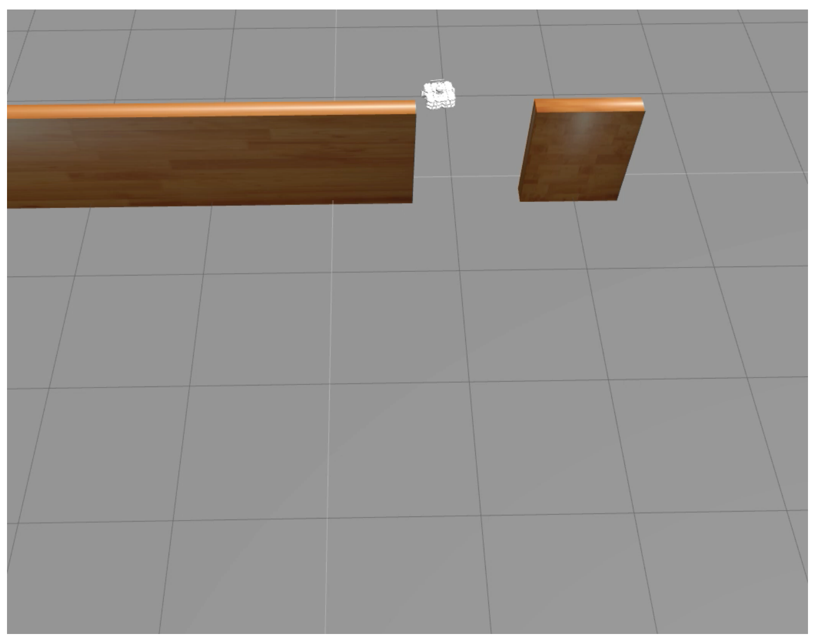
1. The robot will move forward until the robot is in parallel with the opening in the wall.



2. Once the robot is in parallel with the opening, it will rotate 90º in the direction of the opening.

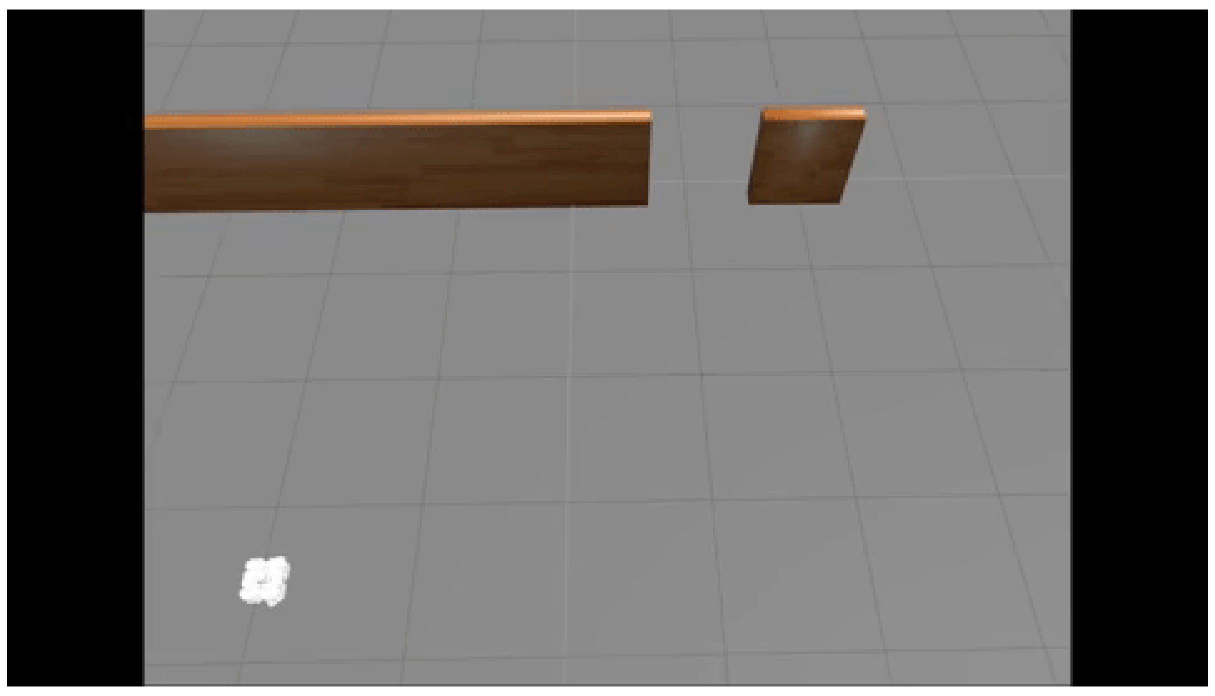


3. Then, the robot will move forward until it goes past the opening (to the other side of the wall).

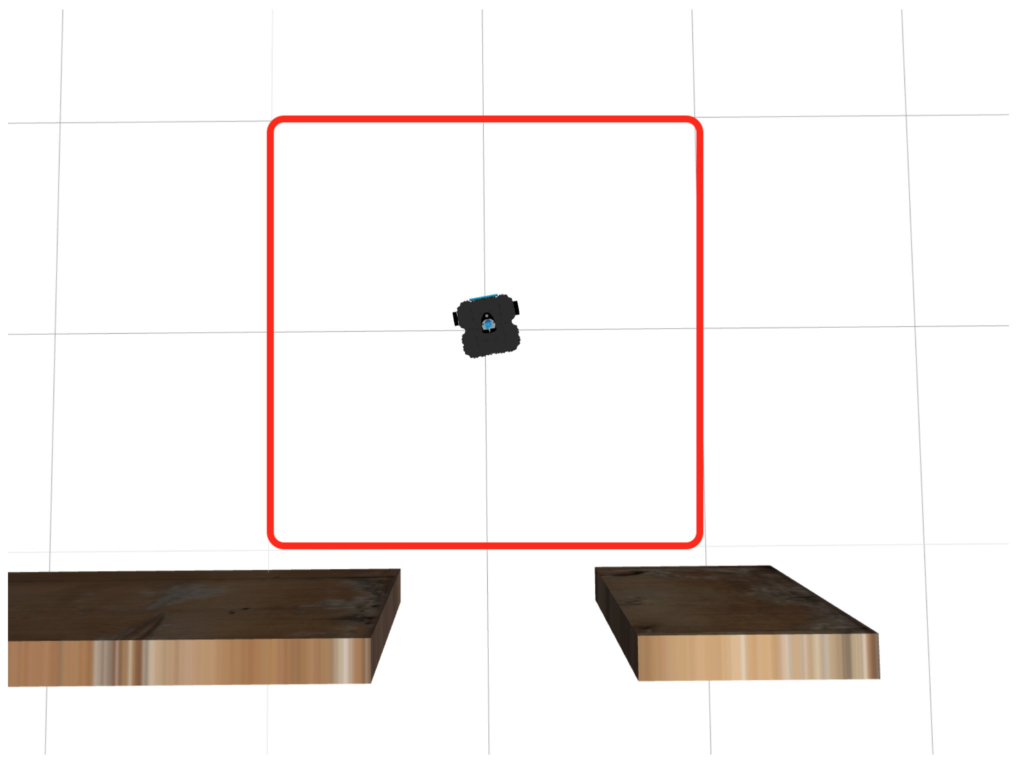


The logic explained above has to result in behavior like the following:

The robot starts moving forward until it is in parallel with the opening in the wall. Then it begins to move forward until it goes past the opening. Finally, the robot stops.



IMPORTANT: The robot final position has to be within the highlighted square:



- Hints -

HINT 1: To perform the 90º rotation using the odometry data, we recommend that you transform the odometry orientation data, which is in quaternions, into euler angles.

To do so, you can use the below function:

In [ ]:

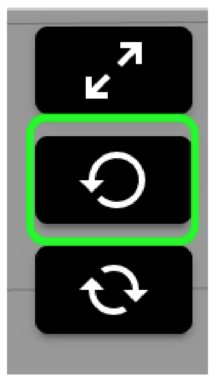
```
import numpy as np

def euler_from_quaternion(self, quaternion):
        """
        Converts quaternion (w in last place) to euler roll, pitch, yaw
        quaternion = [x, y, z, w]
        Below should be replaced when porting for ROS2 Python tf_conversions is done.
        """
        x = quaternion[0]
        y = quaternion[1]
        z = quaternion[2]
        w = quaternion[3]

        sinr_cosp = 2 * (w * x + y * z)
        cosr_cosp = 1 - 2 * (x * x + y * y)
        roll = np.arctan2(sinr_cosp, cosr_cosp)

        sinp = 2 * (w * y - z * x)
        pitch = np.arcsin(sinp)

        siny_cosp = 2 * (w * z + x * y)
        cosy_cosp = 1 - 2 * (y * y + z * z)
        yaw = np.arctan2(siny_cosp, cosy_cosp)

        return roll, pitch, yaw
```

HINT 2: To rotate 90º, the equivalent value in Euler angles is 1.57 radians.

- End of Hints -

- Notes -

For testing purposes, remember that you can send the robot back to its initial position by clicking on the `Reset the simulation` button:



Or you can also do it directly through the command line with the following command:

In [ ]:

```
ros2 service call /reset_world std_srvs/srv/Empty {}
```

- End of Notes -

Specifications

- The name of the package where you will place all the code related to the quiz will be **topics_quiz**.
- The name of the launch file that will start your program will be **topics_quiz.launch.py**.
- The name of the ROS2 node that your program will launch will be **topics_quiz_node**. *This is the node name you **must** specify in the launch file, regardless of what you specify in the script.*

Grading Guide

The following will be checked, in order. *If a step fails, the steps following are skipped.*

1. Does the package exist?
2. Did the package compile successfully?
3. Can the package be launched with the launch file?
4. Was the subscriber created as specified?
5. Was the robot publishing to `/cmd_vel` ?
6. Did the robot go through the opening?

The grader will provide as much feedback on any failed step so you can make corrections where necessary.

Quiz Correction

When you have finished the quiz, you can correct it to get a mark. For that, click on the button at the top of this notebook.

Final Mark

If you fail the quiz or you do not get the desired mark, do not get frustrated! You will have the chance to retake the quiz to improve your score.

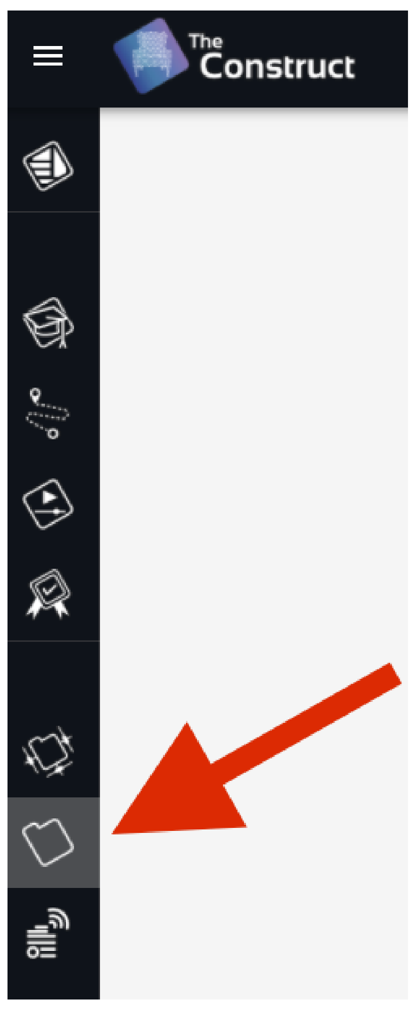
It is now time that you start the project for this course!

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

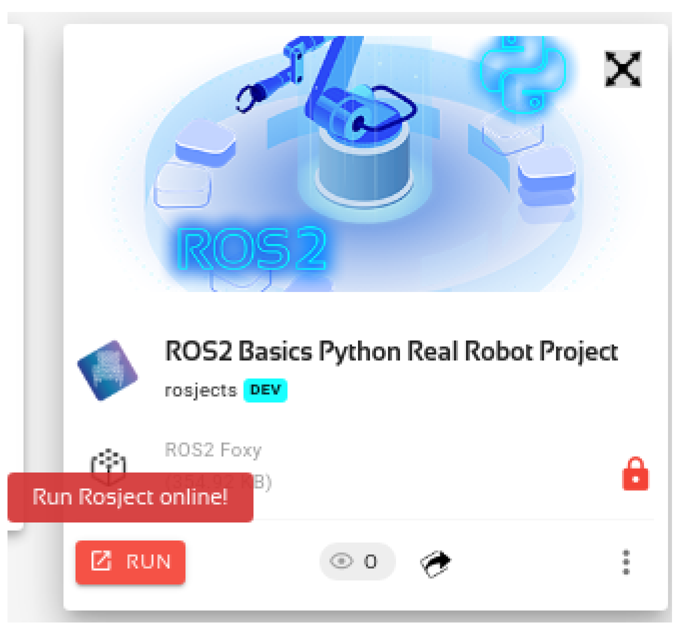
The ROSDS is included with your subscription and is integrated inside The Construct. So no extra effort needs to be made by you. Well, you will need to expend some extra learning effort! But that is why you are here!

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

1. **Copy the project rosject** to your ROSDS area (see instructions below).
2. Once you have it, go to the *My Rosjects* area in The Construct



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



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

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

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

English proofread