ROS2 Basics in 5 Days (Python)

Unit 4 Understanding ROS2 Services

- Summary -

Estimated time to completion: **4.5 hours**

What will you learn with this unit?

- Understand a Service
- Basic Service commands
- Understand a Service Client
- Understand a Service Server
- Custom interfaces

- End of Summary -

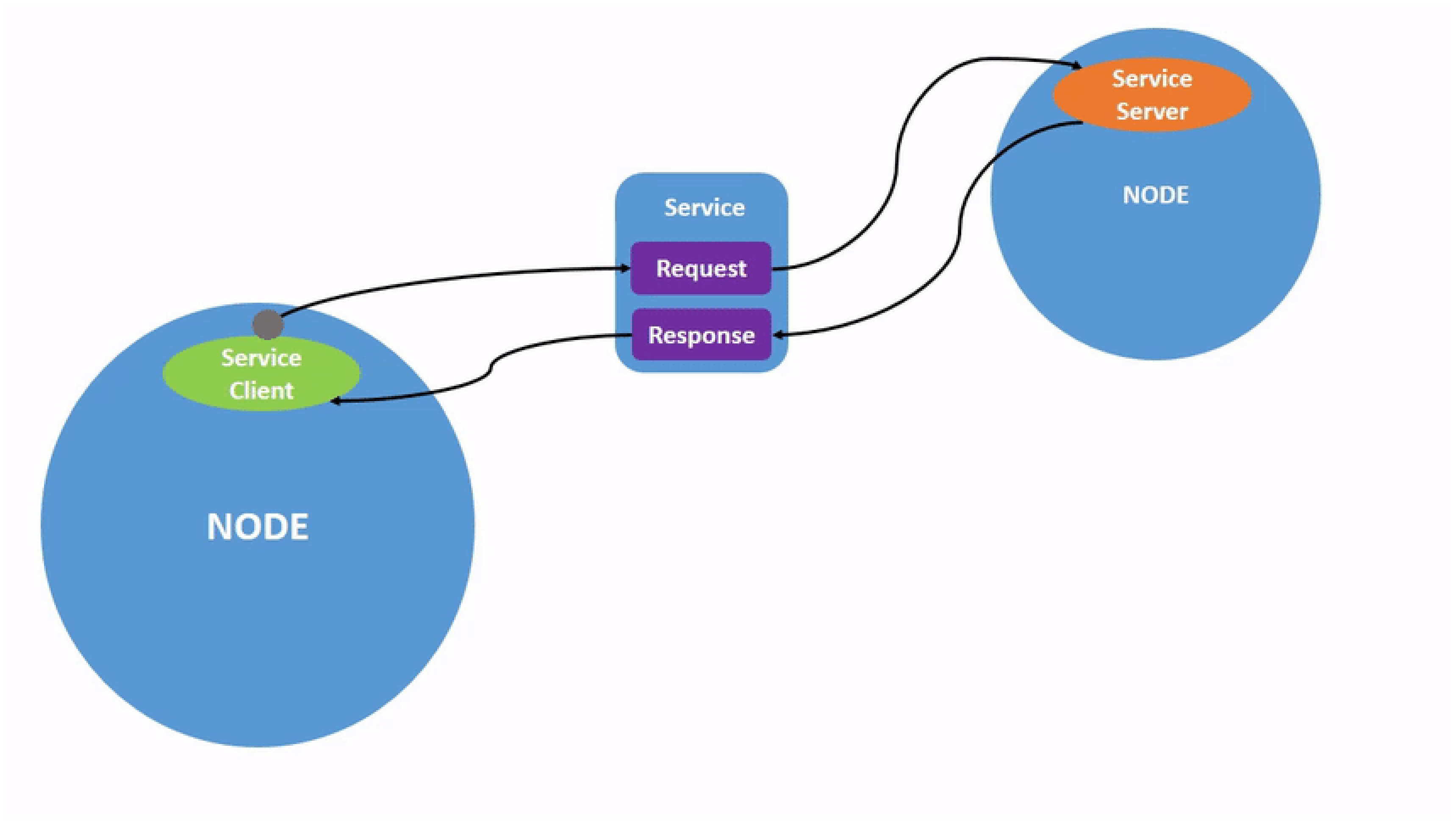
4.1 What is a Service in ROS2?

Like Topics, Services are also a method of communication between nodes for ROS2. So, to understand them better, compare them with something you already know. Topics use the **publisher-subscriber** model. On the other hand, Services use a **call-response** model. As you saw, you can subscribe a node to a Topic to receive particular information with continuous updates. At the same time, a Service only provides data when called explicitly by a Client.

Understand all this in a better way through the following example. Think about a face recognition system. The best approach will be to provide it using a Service. Then, your ROS2 program will call that Service (**send a request to the Service**) every time it needs the name of the person your recognition system has in front of it. Then, the Service will **return a response** providing the person's name.

And why is it better to use a Service in this case? Well, because you do not need to constantly run your face detection node when there is nobody around the robot. It would be a massive waste of resources. You only need to run it when there is a person in front of the robot.

Working with Services, you will have two sides: **Clients** and **Servers**. You can have multiple Clients using the same Service Server, but you can only have one Server for one Service.



The animated image is taken from the official ROS2 documentation

4.2 Basic Service Commands

To start working with Services, you must review basic commands. These are similar to the commands you saw in the Topics module.

- Example 4.1 -

The first command you will review is the `ros2 service list`. This command will list all the Services currently available in your ROS2 system. To do that, type the following command in Shell #1.

Execute in Shell #1

In [ ]:

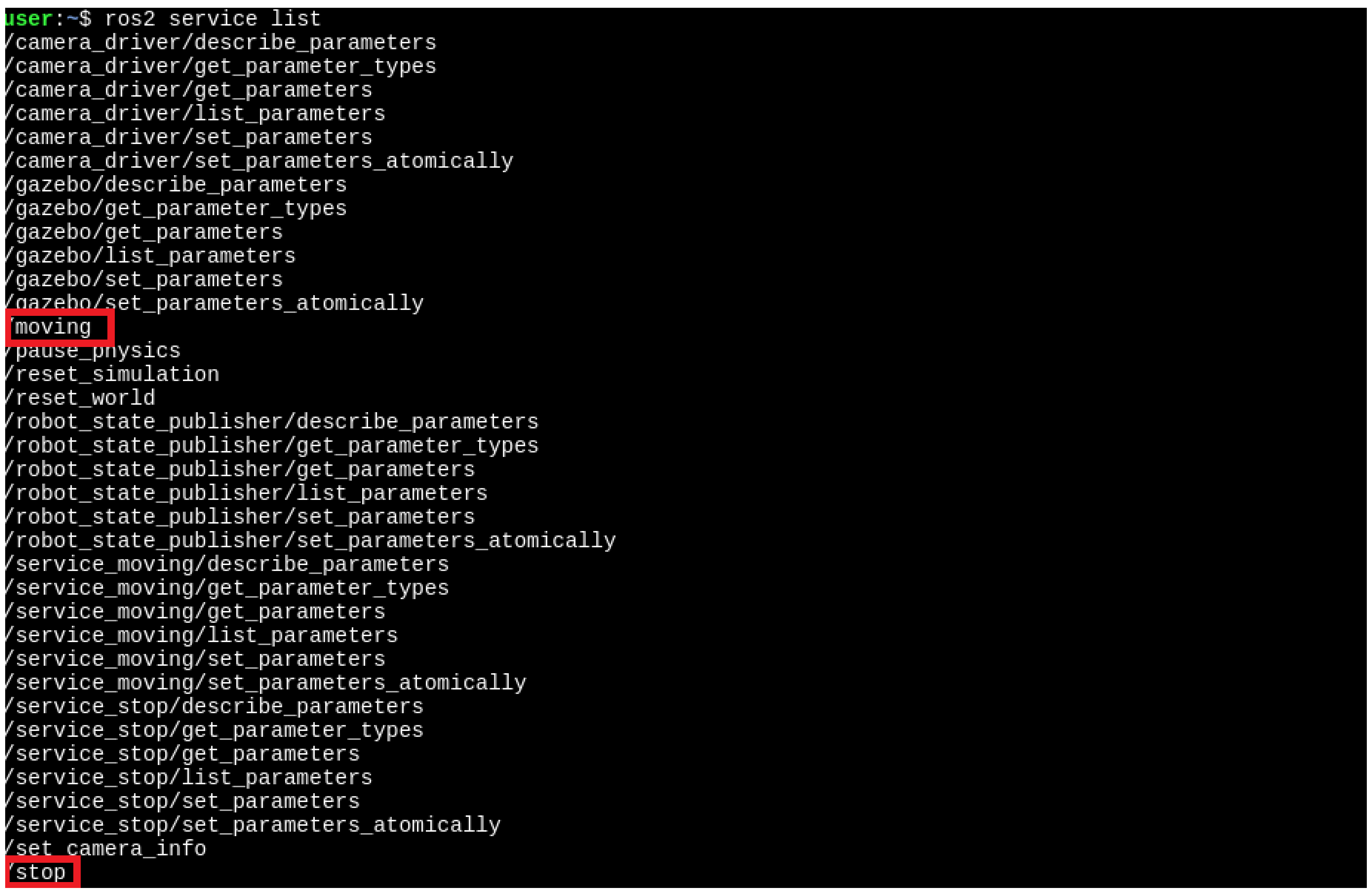
```
source /opt/ros/humble/setup.sh
```

In [ ]:

```
ros2 service list
```

Of the Services that you can see listed on your terminal, of interest are the following:

Shell #1 Output



```
user:~$ ros2 service list
/camera_driver/describe_parameters
/camera_driver/get_parameter_types
/camera_driver/get_parameters
/camera_driver/list_parameters
/camera_driver/set_parameters
/camera_driver/set_parameters_atomically
/gazebo/describe_parameters
/gazebo/get_parameter_types
/gazebo/get_parameters
/gazebo/list_parameters
/gazebo/set_parameters
/gazebo/set_parameters_atomically
/moving
/pause_physics
/reset_simulation
/reset_world
/robot_state_publisher/describe_parameters
/robot_state_publisher/get_parameter_types
/robot_state_publisher/get_parameters
/robot_state_publisher/list_parameters
/robot_state_publisher/set_parameters
/robot_state_publisher/set_parameters_atomically
/service_moving/describe_parameters
/service_moving/get_parameter_types
/service_moving/get_parameters
/service_moving/list_parameters
/service_moving/set_parameters
/service_moving/set_parameters_atomically
/service_stop/describe_parameters
/service_stop/get_parameter_types
/service_stop/get_parameters
/service_stop/list_parameters
/service_stop/set_parameters
/service_stop/set_parameters_atomically
/set_camera_info
/stop
```

In this section, you will work with the `/moving` and `/stop` Services inside the red boxes.

- End of Example 4.1 -

Now that you know what Services are available, you will see how to call them.

- Example 4.2 -

The command you will see now is the `ros2 service call`. This command is used to call a Service (send a request). The structure of the command is as follows:

In [ ]:

```
ros2 service call <service_name> <service_type> <value>
```

So, you need the type of Service to call it. Of course, you also need to pass it a value. What can you do to find which type of Service it is? Is there a command that does it for you?

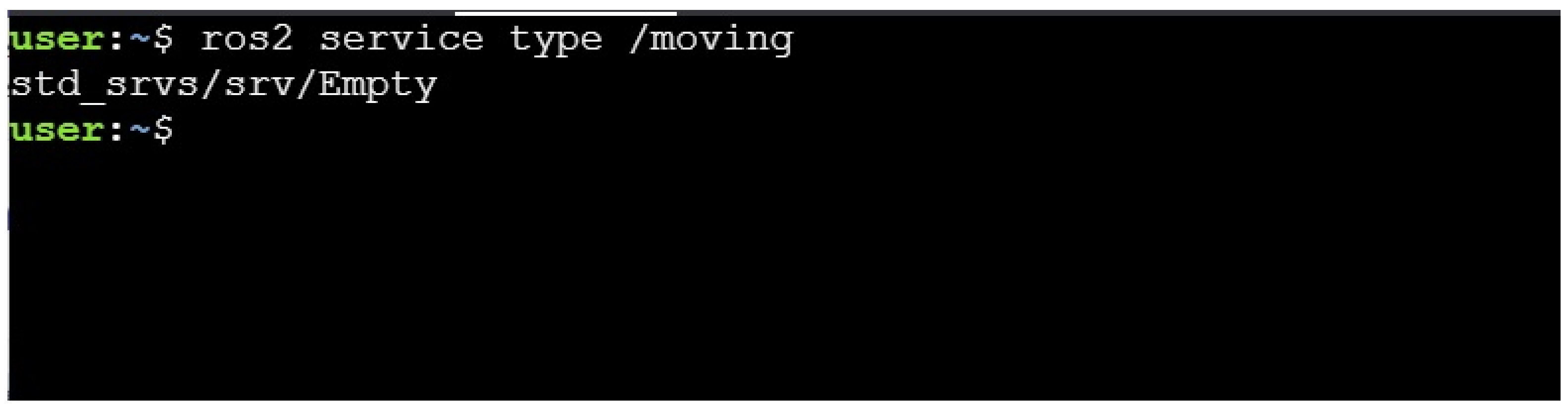
The answer is yes! Indeed, there is a command to know what type a Service uses. This is the `ros2 service type`. See how it works and type the following in your Shell #1.

Execute in Shell #1

In [ ]:

```
ros2 service type /moving
```

Shell #1 Output



```
user:~$ ros2 service type /moving
std_srvs/srv/Empty
user:~$
```

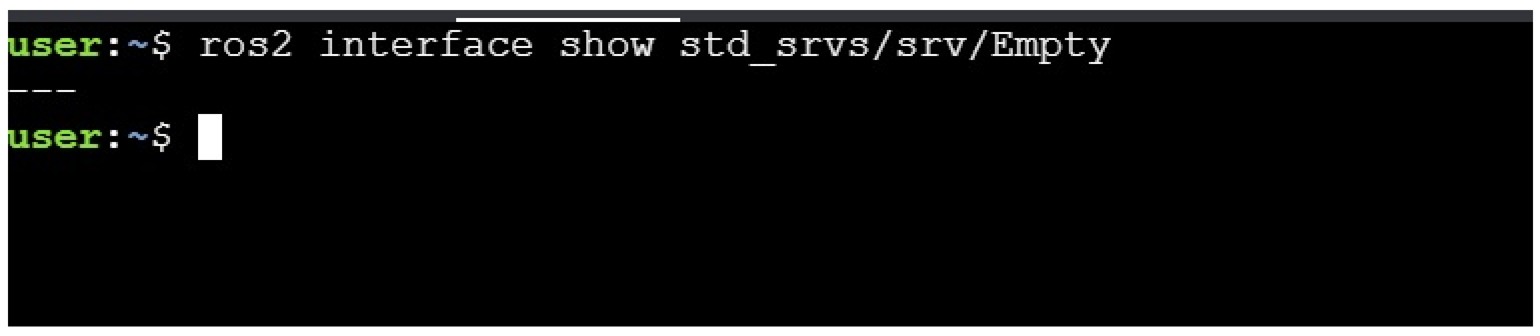
The output tells you that the `/moving` Service works with the `std_srvs/srv/Empty` **Service type interface**. As you may remember from the previous unit, you can use the interface commands you learned in the previous unit to get more information.

Execute in Shell #1

In [ ]:

```
ros2 interface show std_srvs/srv/Empty
```

Shell #1 Output



```
user:~$ ros2 interface show std_srvs/srv/Empty
---
user:~$
```

Alright, so you can see that this type does not have any data. Do not worry. This is normal! This is because you are dealing with the `Empty` Service type, which does not contain any data.

If you remember to use the `ros2 service call` command, you need the **Service's name**, the **the type Service's type**, and a **value** to send (which you now know as **request data**). However, as you see from the name of the Service you are using and the result you see in your terminal, in this case, you do not need to send a value when calling the Service.

- Notes -

Note that the Service type contains three dashes:

```
---
```

This is VERY IMPORTANT in Service types because it separates the **request** part of the message from the **response**. It is structured like this:

```
<request>
---
<response>
```

In this case, there is no data because you are dealing with the `Empty` Service type. But most Service types will have a defined request and response. For instance, check another Service type:

Execute in Shell #1

In [ ]:

```
ros2 interface show std_srvs/srv/SetBool
```

Shell #1 Output

```
bool data # e.g., for hardware enabling/disabling
---
bool success   # indicates a successful run of the triggered service
string message # informational, e.g., for error messages
```

- End of Notes -

Okay. With all that said, let us get going. The interesting part comes now that you know how to interact with the Service.

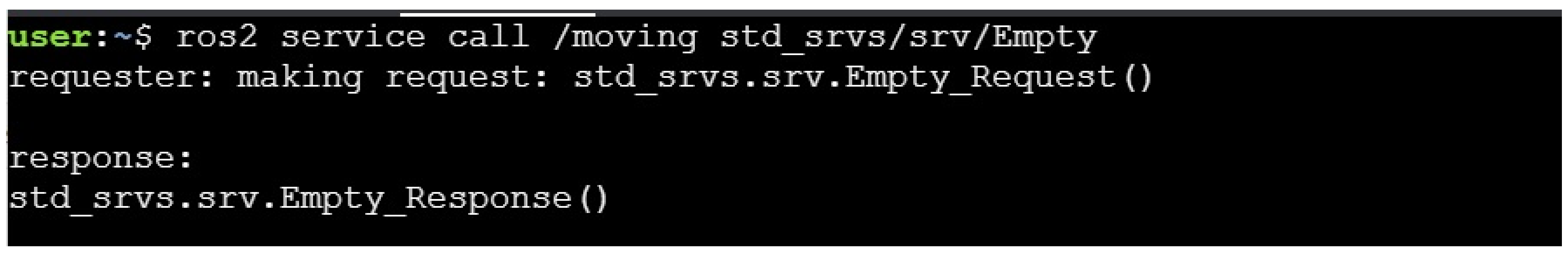
Now, try the `ros2 service call` command.

Execute in Shell #1

In [ ]:

```
ros2 service call /moving std_srvs/srv/Empty
```

Shell #1 Output

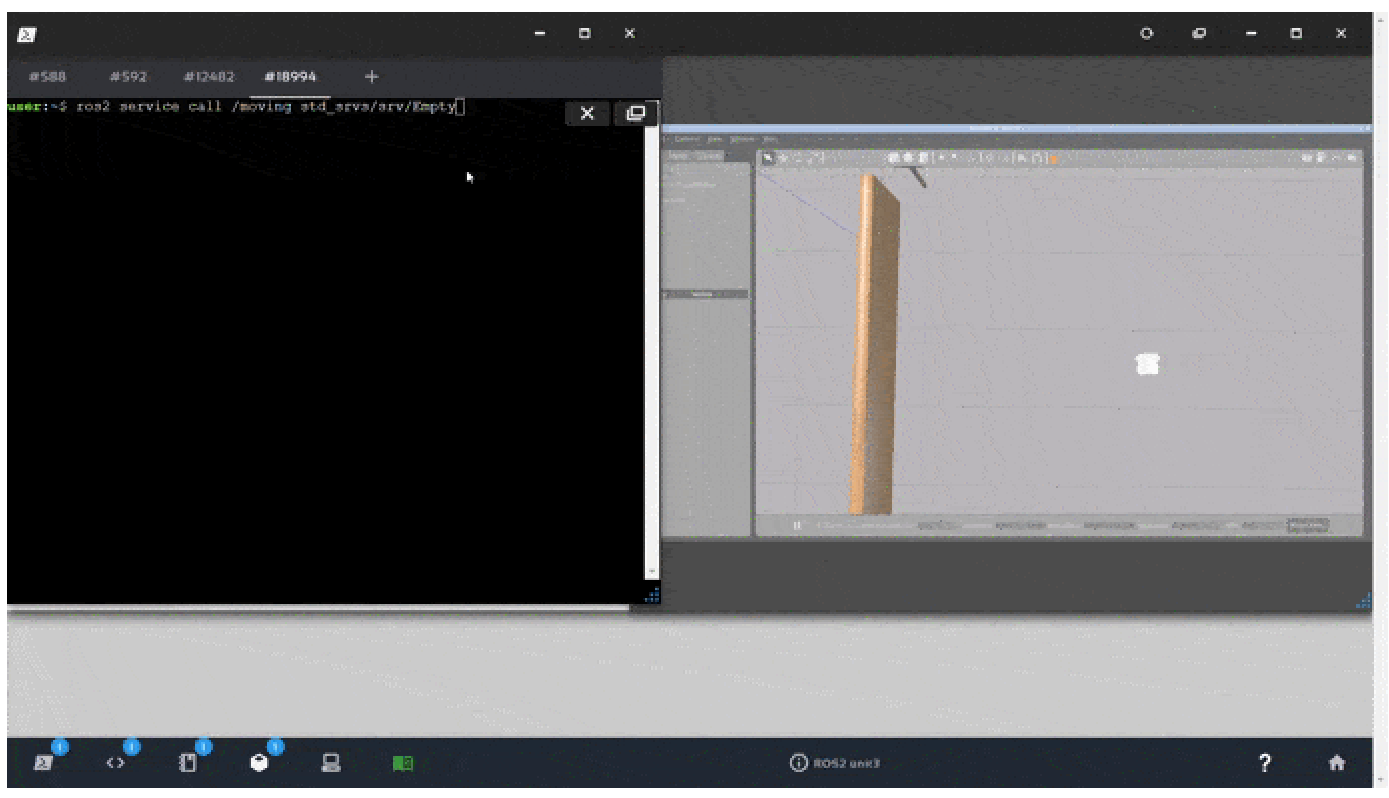


```
user:~$ ros2 service call /moving std_srvs/srv/Empty
requester: making request: std_srvs.srv.Empty_Request()

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

In summary, the last command you executed does the following:

- Look for a Service named `/moving`
- Create a Client along with a request and send it to the corresponding Service
- Wait for the Server response and print it



```
user:~$ ros2 service call /moving std_srvs/srv/Empty
```

You will notice in the Gazebo simulation that the robot has started to move. How do you make it stop? Do you remember there was another Service called `/stop`? Try to use it.

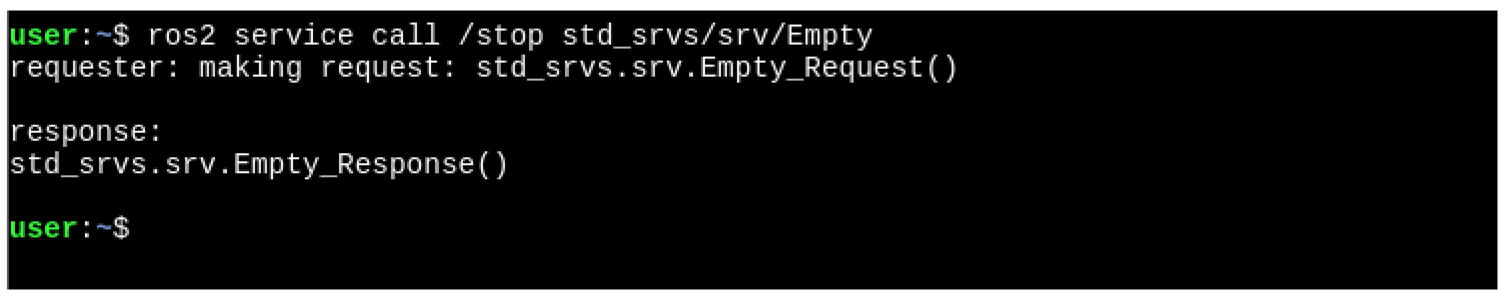
For that, I need you to type the following command:

Execute in Shell #1

In [ ]:

```
ros2 service call /stop std_srvs/srv/Empty
```

Shell #1 Output

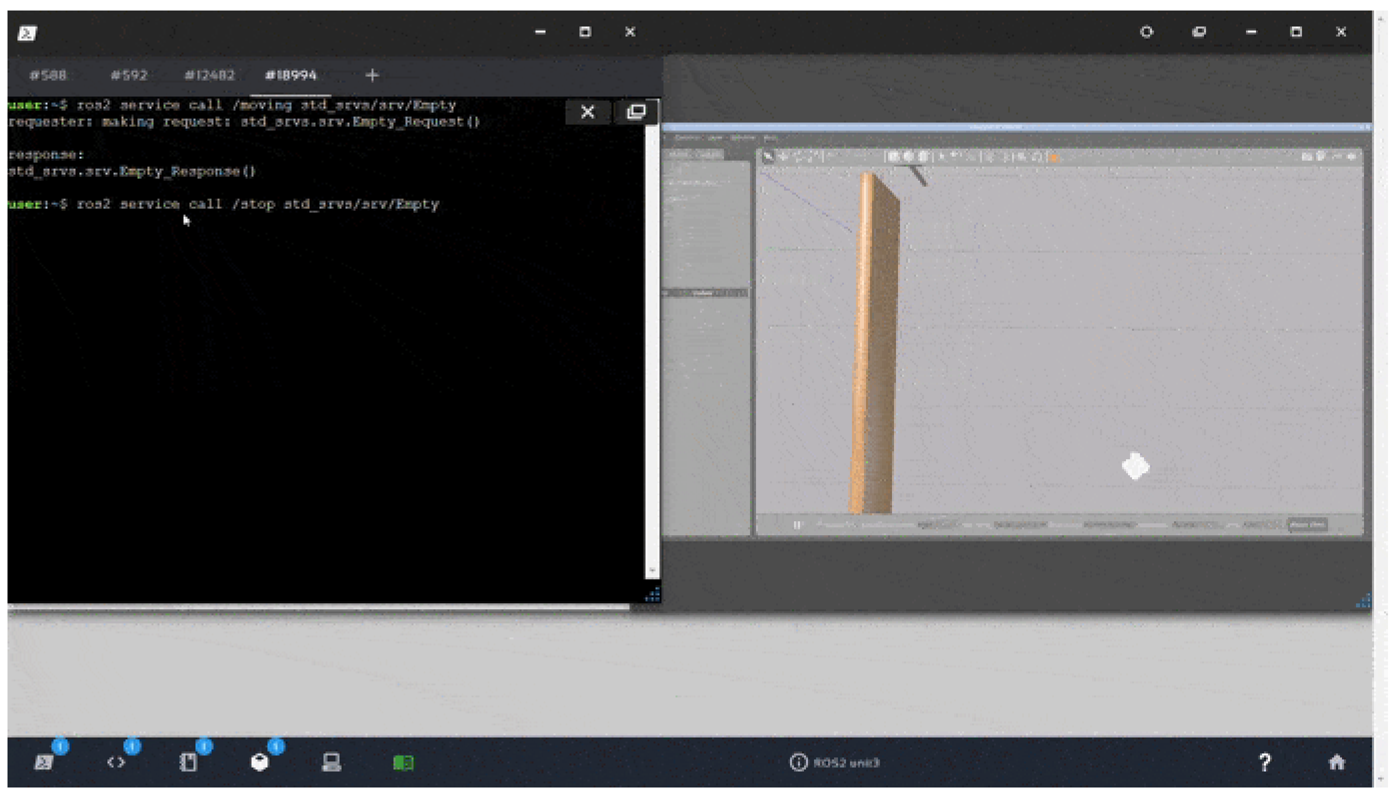


```
user:~$ ros2 service call /stop std_srvs/srv/Empty
requester: making request: std_srvs.srv.Empty_Request()

response:
std_srvs.srv.Empty_Response()

user:~$
```

You will see the same result in the terminal as when you called the `/moving` Service. The difference will be in the Gazebo simulation because the robot will stop moving.



```
user:~$ ros2 service call /moving std_srvs/srv/Empty
requester: making request: std_srvs.srv.Empty_Request()

response:
std_srvs.srv.Empty_Response()

user:~$ ros2 service call /stop std_srvs/srv/Empty
```

- Notes -

As you may have noticed, for the `/stop` Service, you did not stop to analyze the type of Service involved because it is the same as `/moving`. This is not something that will always happen. When working with a new Service, always make sure you know how to interact correctly with it using the `ros2 service type` command.

- End of Notes -

- End of Example 4.2 -

As you have seen, you can use the **`ros2 service call`** command to emulate a Client without creating a new program, which is especially useful for testing a given Service. However, as your ROS2 application gets bigger and takes shape, you will need to implement a Client that can interact with a Service from their nodes, and this is what you will see in the next sections.

4.3 Create a Service Client

With everything you have learned, you are ready to write your first Client and make it interact with one of the Services you saw in the previous section.

- Example 4.3 -

1.- Create a new package named **client_pkg** on your `~/ros2_ws/src` directory.

Execute in Shell #1

In [ ]:

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

In [ ]:

```
ros2 pkg create client_pkg --build-type ament_python --dependencies rclpy std_srvs
```

As you remember, you will import the `std_srvs` interface to interact with the `/moving` and `/stop` Services that you saw before.

2.- Create a new file named **service_client.py** inside the **client_pkg** folder you created in the previous step.

Inside the file you just created, write the following code:

service_client.py

In [ ]:

```
# import the empty module from std_servs Service interface
from std_srvs.srv import Empty
# import the ROS2 Python client libraries
import rclpy
from rclpy.node import Node


class ClientAsync(Node):

    def __init__(self):
        # Here you have the class constructor

        # call the class constructor to initialize the node as service_client
        super().__init__('service_client')
        # create the Service Client object
        # defines the name and type of the Service Server you will work with.
        self.client = self.create_client(Empty, 'moving')
        # checks once per second if a Service matching the type and name of the Client is available.
        while not self.client.wait_for_service(timeout_sec=1.0):
            # if it is not available, a message is displayed
            self.get_logger().info('service not available, waiting again...')

        # create an Empty request
        self.req = Empty.Request()


    def send_request(self):

        # send the request
        self.future = self.client.call_async(self.req)


def main(args=None):
    # initialize the ROS communication
    rclpy.init(args=args)
    # declare the node constructor
    client = ClientAsync()
    # run the send_request() method
    client.send_request()

    while rclpy.ok():
        # pause the program execution, waits for a request to kill the node (ctrl+c)
        rclpy.spin_once(client)
        if client.future.done():
            try:
                # checks the future for a response from the Service
                # while the system is running.
                # If the Service has sent a response, the result will be written
                # to a log message.
                response = client.future.result()
            except Exception as e:
                # Display the message on the console
                client.get_logger().info(
                    'Service call failed %r' % (e,))
            else:
                # Display the message on the console
                client.get_logger().info(
                    'the robot is moving' )
            break

    client.destroy_node()
    # shutdown the ROS communication
    rclpy.shutdown()


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

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

Execute in Shell #1

In [ ]:

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

In [ ]:

```
mkdir launch
```

In [ ]:

```
cd launch
```

In [ ]:

```
touch service_client_launch_file.launch.py
```

In [ ]:

```
chmod +x service_client_launch_file.launch.py
```

Write the necessary code to launch the executable files of the **service_client** script.

service_client_launch_file.launch.py

In [ ]:

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

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

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

setup.py

In [ ]:

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

package_name = 'client_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': [
        'service_client = client_pkg.service_client:main',
        ],
    },
)
```

5.- Compile your package as you know how to.

Execute in Shell #1

In [ ]:

```
cd ~/ros2_ws
```

In [ ]:

```
colcon build --packages-select client_pkg
```

In [ ]:

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

7.- Finally, launch the Service Server node on your shell.

Execute in Shell #1

In [ ]:

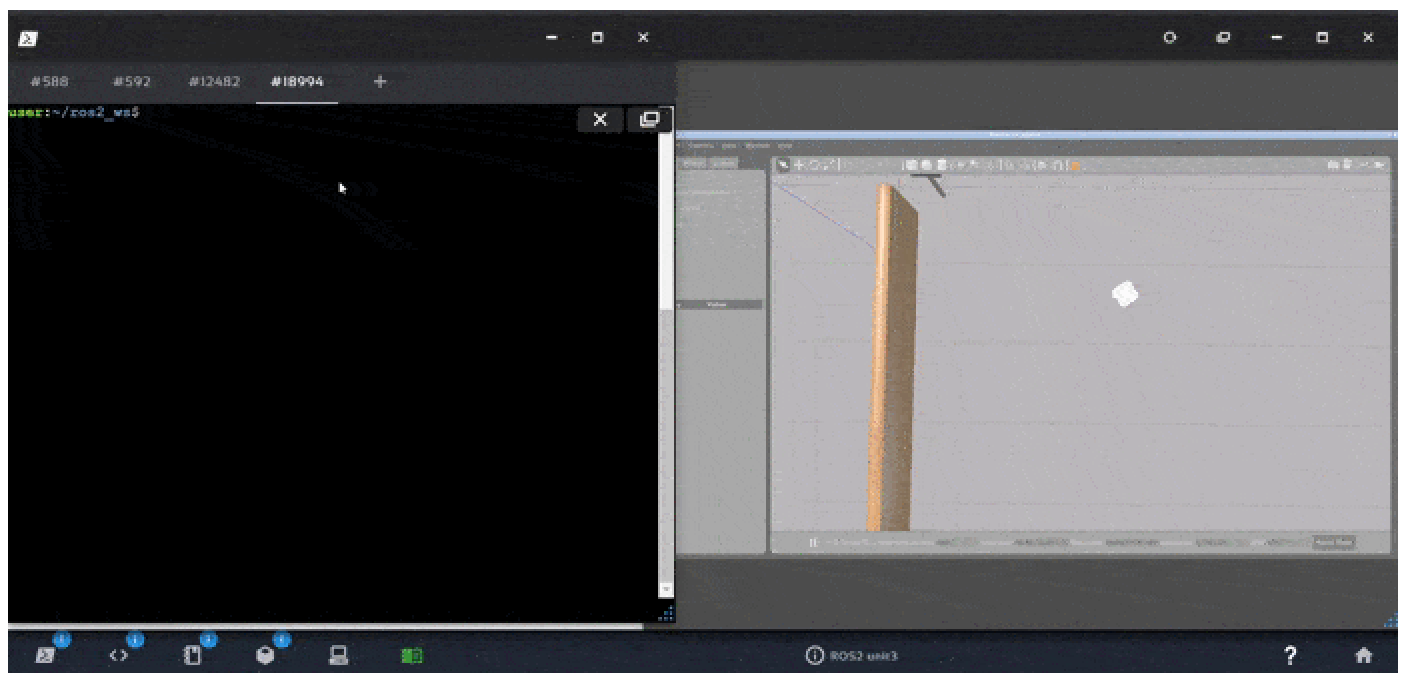
```
ros2 launch client_pkg service_client_launch_file.launch.py
```

You should get a result similar to the following.

Shell #1 Output

```
[INFO] [launch]: All log files can be found below /home/user/.ros/log/2022-08-23-14-34-41-082469-3_xterm-2690
[INFO] [launch]: Default logging verbosity is set to INFO
[INFO] [service_client-1]: process started with pid [2692]
[service_client-1] [INFO] [1661265282.189407377] [service_client]: the robot is moving
[INFO] [service_client-1]: process has finished cleanly [pid 2692]
```

In the simulation, you will see the robot start moving:



You have created a Client node that interacts with a Server node. It seems that you are making progress in Services. Consider that the Client you have just created will only be used to call the `/moving` Service and start the movement of the TurtleBot. To stop the movement, complete the following exercise.

- End of Example 4.3 -

4.3.1 Code Review

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

This is the line where you create the Client:

In [ ]:

```python
self.client = self.create_client(Empty, 'moving')
```

You can see that you create a Client that uses the **`Empty`** Service type and connects to a Service named **`/moving`** .

This `while` loop is used to ensure that the Service Server (in this case, `/moving`) is up and running:

In [ ]:

```python
while not self.client.wait_for_service(timeout_sec=1.0):
```

This line is also very important:

In [ ]:

```
self.future = self.client.call_async(self.req)
```

Send an **asynchronous request** to the Service Server using the `call_async()` method. Then, store the response from the Server in the variable `self.future`. This `future` variable is also very important. After making the request, the Server will immediately return `future`, which indicates whether the call and response are finished (but it does not contain the value of the response itself). You will read more on this in the next section of this chapter.

Finally, you have the `main` loop:

In [ ]:

```
while rclpy.ok():
    # pause the program execution, waits for a request to kill the node (ctrl+c)
    rclpy.spin_once(client)
    if client.future.done():
        try:
            # checks the future for a response from the Service
            # while the system is running.
            # If the Service has sent a response, the result will be written
            # to a log message.
            response = client.future.result()
        except Exception as e:
            # Display the message on the console
            client.get_logger().info(
                'Service call failed %r' % (e,))
        else:
            # Display the message on the console
            client.get_logger().info(
                'the robot is moving' )
        break
```

Here, you keep checking (while the program is running, `rclpy.ok()`) if the Service response is finished:

In [ ]:

```
if client.future.done():
```

If it is finished, you then get the value of the Server response:

In [ ]:

```
response = client.future.result()
```

In this case, since you are working with an `Empty` response, you do nothing else.

While you wait for the Service response to finish, you are printing messages to the node's log:

In [ ]:

```
client.get_logger().info('the robot is moving' )
```

Finally, it is also worth mentioning the following line of code:

In [ ]:

```
rclpy.spin_once(client)
```

This will spin the `client` node one time. It is similar to `rclpy.spin` (that you already saw), but instead of spinning the node indefinitely, it will only spin it once.

- Exercise 4.1 -

Create a new Service Client, for example, the one you wrote in Example 4.3. The Client that you create must interact with the `/stop` Service to stop the movement of the TurtleBot in the simulation.

- End of Exercise 4.1 -

- Solution for Exercise 4.1 -

1.- Create a new file named **stop_client.py** inside the **client_pkg** folder you created in the previous section.

Inside the file you just created, write the following code:

stop_client.py

In [ ]:

```
# import the Empty module from std_servs Service interface
from std_srvs.srv import Empty
# import the ROS2 Python client libraries
import rclpy
from rclpy.node import Node


class ClientAsync(Node):

    def __init__(self):
        # Here you have the class constructor

        # call the class constructor to initialize the node as stop_client
        super().__init__('stop_client')
        # create the Service Client object
        # defines the name and type of the Service Server you will work with.
        self.client = self.create_client(Empty, 'stop')
        # checks once per second if a Service matching the type and name of the Client is available.
        while not self.client.wait_for_service(timeout_sec=1.0):
            # if it is not available, a message is displayed
            self.get_logger().info('service not available, waiting again...')

        # create an Empty request
        self.req = Empty.Request()


    def send_request(self):

        # send the request
        self.future = self.client.call_async(self.req)


def main(args=None):
    # initialize the ROS communication
    rclpy.init(args=args)
    # declare the node constructor
    client = ClientAsync()
    # run the send_request() method
    client.send_request()

    while rclpy.ok():
        # pause the program execution, waits for a request to kill the node (ctrl+c)
        rclpy.spin_once(client)
        if client.future.done():
            try:
                # checks the future for a response from the Service
                # while the system is running.
                # If the Service has sent a response, the result will be written
                # to a log message.
                response = client.future.result()
            except Exception as e:
                # Display the message on the console
                client.get_logger().info(
                    'Service call failed %r' % (e,))
            else:
                # Display the message on the console
                client.get_logger().info(
                    'the robot is stopped' )
            break

    client.destroy_node()
    # shutdown the ROS communication
    rclpy.shutdown()


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

2.- Create a launch file named **stop_client_launch_file.launch.py** to launch the node.

Execute in Shell #2

In [ ]:

```
cd ~/ros2_ws/src/client_pkg/launch
```

In [ ]:

```
touch stop_client_launch_file.launch.py
```

In [ ]:

```
chmod +x stop_client_launch_file.launch.py
```

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

stop_client_launch_file.launch.py

In [ ]:

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

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

3.- Modify the setup.py to add the entry points to the executable for the **stop_client** script.

setup.py

In [ ]:

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

package_name = 'client_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': [
        'service_client = client_pkg.service_client:main',
        'stop_client = client_pkg.stop_client:main'
        ],
    },
)
```

5.- Compile your package as you know how to.

Execute in Shell #2

In [ ]:

```
cd ~/ros2_ws
```

In [ ]:

```
colcon build --packages-select client_pkg
```

In [ ]:

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

7.- Finally, launch the Service Server node on your shell.

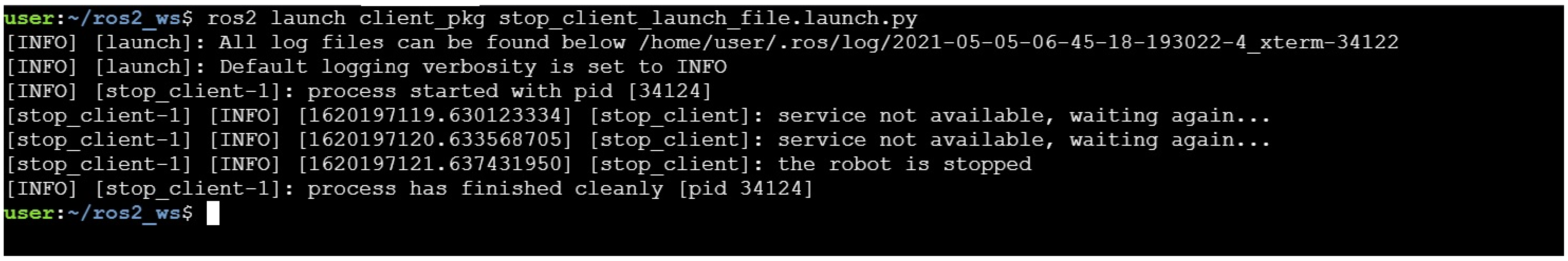
Execute in Shell #2

In [ ]:

```
ros2 launch client_pkg stop_client_launch_file.launch.py
```

You should get a result similar to the following.

Shell #2 Output



```
user:~/ros2_ws$ ros2 launch client_pkg stop_client_launch_file.launch.py
[INFO] [launch]: All log files can be found below /home/user/.ros/log/2021-05-05-06-45-18-193022-4_xterm-34122
[INFO] [launch]: Default logging verbosity is set to INFO
[INFO] [stop_client-1]: process started with pid [34124]
[stop_client-1] [INFO] [1620197119.630123334] [stop_client]: service not available, waiting again...
[stop_client-1] [INFO] [1620197120.633568705] [stop_client]: service not available, waiting again...
[stop_client-1] [INFO] [1620197121.637431950] [stop_client]: the robot is stopped
[INFO] [stop_client-1]: process has finished cleanly [pid 34124]
user:~/ros2_ws$
```

- End of Solution for Exercise 4.1 -

4.4 Synchronous vs. Asynchronous Service Clients in ROS2

You will now review a slightly more complicated subject matter. However, do not worry about it. I will try to make it simple.

In the previous section, you may have noticed that when you call the Service (send a request), you wait until the Server returns a response message. However, while waiting for the response to be finished, you can keep executing other tasks in your node (like sending messages to the log).

So, a Service Client node will start a calling thread to send a request to a Service Server node. What is interesting now is how this call thread will be established. There are two calls a Service Client can make: **Asynchronous** calls and **Synchronous** calls.

In the previous section, you used an **asynchronous** call using the `call_async()` method:

In [ ]:

```
self.future = self.client.call_async(self.req)
```

After sending a request to a Service, an asynchronous client will immediately return `future`, a value that tells if the call and response are complete (not the value of the response itself). At any time, the returned future can be queried for a response.

In [ ]:

```
if client.future.done():
```

In a few words, sending a request using an asynchronous client does not block anything. Then, for instance, you can use a spin function and check for features in the same thread.

Asynchronous calls are entirely safe and the recommended method of calling Services so that they can be made from anywhere without running the risk of blocking other ROS2 processes.

Here is an example of the basic structure of an asynchronous client:

In [ ]:

```
# initialize the ROS communication
rclpy.init(args=args)
# declare the node constructor
client = ClientAsync()
# run the send_request() method
client.send_request()
while rclpy.ok():
        # pause the program execution, waits for a request to kill the node (ctrl+c)
        rclpy.spin_once(client)
        if client.future.done():
            try:
                # checks the future for a response from the Service
                # while the system is running.
                # If the Service has sent a response, the result will be written
                # to a log message.
                response = client.future.result()
            except Exception as e:
                # Display the message on the console
                client.get_logger().info(
                    'Service call failed %r' % (e,))
            else:
                # Display the message on the console
                client.get_logger().info(
                    'Pretty message' )
            break

client.destroy_node()
# shutdown the ROS communication
rclpy.shutdown()
```

On the other hand, a **synchronous client will block the calling thread when sending a request until a response has been received from the server. After that, nothing else can happen on that thread during the call.** The main problem is that you do not know the time required for the call thread to be completed. This could cause a deadlock, which is the permanent blocking of a set of threads in a concurrent system that compete to communicate with each other.

Here is an example of how a synchronous client establishes the communication thread.

In [ ]:

```
# initialize the ROS communication
rclpy.init(args=args)
# declare the node constructor
client = ClientSync()
# start the communication thread
spin_thread = Thread(target=rclpy.spin, args=(client,))
spin_thread.start()
# run the send_request() method
response = client.send_request()
# Display the message on the console
client.get_logger().info('Pretty message')

minimal_client.destroy_node()
# shutdown the ROS communication
rclpy.shutdown()
```

- Notes -

If you have worked with ROS1, you may remember that Services are synchronous. Now working with ROS2, Services are by default asynchronous. This does not mean that you cannot have synchronous Services if you want, but it is **not recommended**. Anyway, I leave you a link to the documentation if you want to learn more about this: ROS2 Docs

- End of Notes -

4.5 Create a Service Server

Remember, you can make the nodes communicate using Services. You named as **Client** the node that sends a request to a Service. On the other side, the node that will respond to that request will be the **Server** node. As you will remember, at the beginning of this module, you worked with the `/moving` and `/stop` Services. Behind them, there is a Server running.

- Example 4.4 -

Now I will show you the codes behind the servers you worked with during this unit. Do not worry. It is nothing too complex or different from the scripts you saw earlier in this course.

Begin with the following code, which was written to implement the `/moving` Service that you use to move the TurtleBot in the Gazebo simulation.

In [ ]:

```python
# import the Empty module from std_servs Service interface
from std_srvs.srv import Empty
# import the Twist module from geometry_msgs messages interface
from geometry_msgs.msg import Twist
# import the ROS2 Python client libraries
import rclpy
from rclpy.node import Node


class Service(Node):

    def __init__(self):
        # Here you have the class constructor

        # call the class constructor to initialize the node as service_moving
        super().__init__('service_moving')
        # create the Service Server object
        # defines the type, name, and callback function
        self.srv = self.create_service(Empty, 'moving', self.empty_callback)
        # 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
        self.publisher_ = self.create_publisher(Twist, 'cmd_vel', 10)


    def empty_callback(self, request, response):
        # The callback function receives the self-class parameter,
        # received along with two parameters called request and response
        # - receive the data by request
        # - return a result as a response

        # create a Twist message
        msg = Twist()
        # define the linear x-axis velocity of /cmd_vel topic parameter to 0.3
        msg.linear.x = 0.3
        # define the angular z-axis velocity of /cmd_vel topic parameter to 0.3
        msg.angular.z = 0.3
        # Publish the message to the Topic
        self.publisher_.publish(msg)
        # print a pretty message
        self.get_logger().info('RUN ROBOT RUN!')


        # return the response parameter
        return response


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


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

Have a look at the most important parts of the code.

This is the line where the Server is being created:

In [ ]:

```python
self.srv = self.create_service(Empty, 'moving', self.empty_callback)
```

- The Service uses an `Empty` type
- The name of the Service is `moving`
- The Service callback is `empty_callback`

Every time the `/moving` Service receives a request from a Client, the `empty_callback` method is executed.

Also, analyze the following code, which was written to implement the `/stop` Service that you use to stop the TurtleBot movement in the Gazebo simulation. You will not find many other differences than the name under which the Service will be launched, and the speed parameters for the /cmd_vel topic needed to stop the robot.

In [ ]:

```
# import the empty module from std_servs Service interface
from std_srvs.srv import Empty
# import the Twist module from geometry_msgs messages interface
from geometry_msgs.msg import Twist
# import the ROS2 Python client libraries
import rclpy
from rclpy.node import Node


class Service(Node):

    def __init__(self):
        # Here you have the class constructor

        # call the class constructor to initialize the node as service_stop
        super().__init__('service_stop')
        # create the Service server object
        # defines the type, name, and callback function
        self.srv = self.create_service(Empty, 'stop', self.empty_callback)
        # 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
        self.publisher_ = self.create_publisher(Twist, 'cmd_vel', 10)


    def empty_callback(self, request, response):
        # The callback function receives the self-class parameter,
        # received along with two parameters called request and response
        # - receive the data by request
        # - return a result as a response

        # create a Twist message
        msg = Twist()
        # define the linear x-axis velocity of /cmd_vel topic parameter to 0
        msg.linear.x = 0.0
        # define the angular z-axis velocity of /cmd_vel topic parameter to 0
        msg.angular.z = 0.0
        # Publish the message to the topic
        self.publisher_.publish(msg)
        # print a pretty message
        self.get_logger().info('Stop there, Robot!')

        # return the response parameter
        return response


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


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

Okay. Now you know what is behind the Services running in your workspace, at least a few simple ones. Now, demonstrate how much you have learned with the exercise below.

- End of Example 4.4 -

- Exercise 4.2 -

Based on the previous examples, make a Service that executes the following instructions:

- Create a new package and call it **exercise42_pkg** with `rclpy`, `std_msgs`, `sensor_msgs`, `geometry_msgs`, and `std_srvs` as dependencies.
- Use the previous Service Server codes to create a new Service Server that makes the robot turn right (as in the example of the **`/moving`** Service, but the opposite direction). But this time will be a little bit more difficult. You will use the Service **`SetBool`**, part of the **`std_srv`** package. It will work like this:
  - When the request is **true**, the robot turns right.
  - When the input is **false**, the robot stops!
- Create the launch file to start your node.
- Test your Service by calling it using the **`ros2 service call`** command.

- Notes -

Remember, you can get information about the **`SetBool`** Service type with the following command:

In [ ]:

```
ros2 interface show std_srvs/srv/SetBool
```

- End of Notes -

- End of Exercise 4.2 -

- Solution for Exercise 4.2 -

1.- Create a new package named **exercise42_pkg** on your `~/ros2_ws/src` directory and add the necessary dependencies.

Execute in Shell #1

In [ ]:

```
ros2 pkg create exercise42_pkg --build-type ament_python --dependencies rclpy std_msgs geometry_msgs sensor_msgs std_srvs
```

2.- Create a new file named **exercise42.py** inside the **exercise42_pkg** folder inside the new package.

In the file you created, copy the following code:

exercise42.py

In [ ]:

```
# import the SetBool module from std_servs Service interface
from std_srvs.srv import SetBool
# import the Twist module from geometry_msgs messages interface
from geometry_msgs.msg import Twist
# import the ROS2 Python client libraries
import rclpy
from rclpy.node import Node


class Service(Node):

    def __init__(self):
        # Here you have the class constructor

        # call the class constructor to initialize the node as service_moving
        super().__init__('service_moving_right')
        # create the Service Server object
        # defines the type, name, and callback function
        self.srv = self.create_service(SetBool, 'moving_right', self.SetBool_callback)

        # 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
        self.publisher_ = self.create_publisher(Twist, 'cmd_vel', 10)
        # create a Twist message
        self.cmd = Twist()


    def SetBool_callback(self, request, response):
        # The callback function receives the self-class parameter,
        # received along with two parameters called request and response
        # - receive the data by request
        # - return a result as a response

        # Publish the message to the topic
        # As you see, the name of the request parameter is data, so do it
        if request.data == True:

            # define the linear x-axis velocity of /cmd_vel topic parameter to 0.3
            self.cmd.linear.x = 0.3
            # define the angular z-axis velocity of /cmd_vel topic parameter to 0.3
            self.cmd.angular.z =-0.3

            self.publisher_.publish(self.cmd)
            # You need a response
            response.success = True
            # You need another response, but this time, SetBool lets you put a String
            response.message = 'MOVING TO THE RIGHT RIGHT RIGHT!'

        if request.data == False:

            self.cmd.linear.x = 0.0
            # define the angular z-axis velocity of /cmd_vel topic parameter to 0.3
            self.cmd.angular.z =0.0

            self.publisher_.publish(self.cmd)
            response.success = False

            response.message = 'It is time to stop!'

        # return the response parameters
        return response


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


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

3.- Create a launch file named **exercise42_launch_file.launch.py**

Execute in Shell #1

In [ ]:

```
cd ~/ros2_ws/src/exercise42_pkg
```

In [ ]:

```
mkdir launch
```

In [ ]:

```
cd ~/ros2_ws/src/exercise42_pkg/launch
```

In [ ]:

```
touch exercise42_launch_file.launch.py
```

In [ ]:

```
chmod +x exercise42_launch_file.launch.py
```

Inside the new launch file, write the necessary code to launch the executable files of the **exercise42** script.

exercise42_launch_file.launch.py

In [ ]:

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

def generate_launch_description():
    return LaunchDescription([
        Node(
            package='exercise42_pkg',
            executable='exercise42',
            output='screen'),
    ])
```

4.- Modify the setup.py

setup.py

In [ ]:

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

package_name = 'exercise42_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': [
            'exercise42 = exercise42_pkg.exercise42:main'
        ],
    },
)
```

5.- Compile your package

Execute in Shell #1

In [ ]:

```
cd ~/ros2_ws
```

In [ ]:

```
colcon build --packages-select exercise42_pkg
```

In [ ]:

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

7.- Launch the Service Server node in your Shell #1.

Execute in Shell #1

In [ ]:

```
ros2 launch exercise42_pkg exercise42_launch_file.launch.py
```

You should get a result similar to the following.

8.- Finally, call the Service Server node on your Shell #2.

Execute in Shell #2

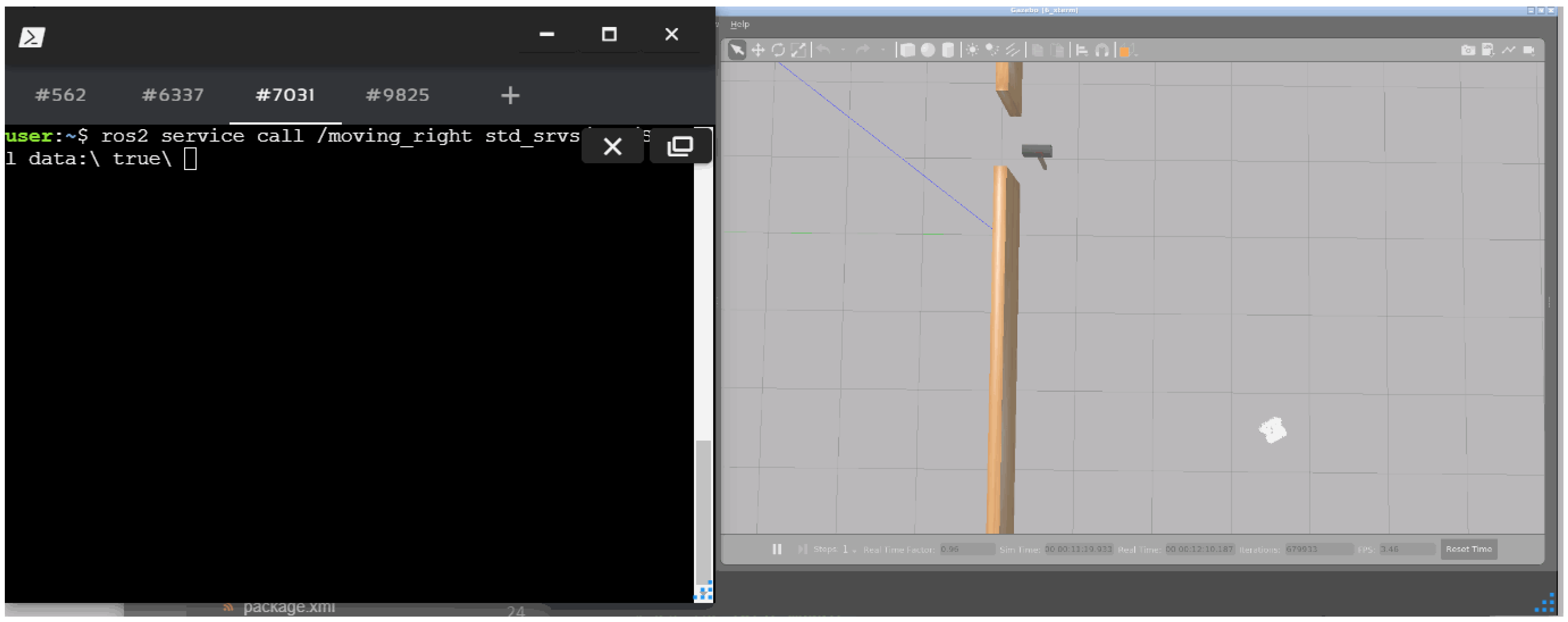
In [ ]:

```
ros2 service call /moving_right std_srvs/srv/SetBool data:\ true
```

And then -

In [ ]:

```
ros2 service call /moving_right std_srvs/srv/SetBool data:\ false
```



```
user:~$ ros2 service call /moving_right std_srvs
l data:\ true\
```

- End of Solution for Exercise 4.2 -

4.6 Custom Service Interface

As you progress along your way with ROS2, you will have more complex projects where you might need to create your customized Service types. We have prepared this section for you so that you will be prepared when that moment arrives. It is a starting point!

- Example 4.5 -

To create a new Service type (srv), complete the following steps:

1. **Create a directory named srv inside your package**
2. **Inside this directory, create a file named `Name_of_your_service_type.srv` (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, create an interface that receives as a request three possible movements: "Turn Right," "Turn Left," and "Stop."

1.- Create a directory srv in your package. For this example, you will use the **custom_interfaces** package that you created in the previous Unit.

Execute in Shell #1

In [ ]:

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

In [ ]:

```
mkdir srv
```

In [ ]:

```
cd srv
```

In [ ]:

```
touch MyCustomServiceMessage.srv
```

2.- MyCustomServiceMessage.srv file must contain this:

In [ ]:

```
string move   # Signal to define the movement
              # "Turn right" to make the robot turn in the right direction.
              # "Turn left" to make the robot turn in the left direction.
              # "Stop" to make the robot stop the movement.

---
bool success  # Did it achieve it?
```

- Notes -

You configured this package in the previous unit, so you will not need to perform the following steps. However, they are summarized below for your reference.

- End of Notes -

3.- In CMakeLists.txt

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

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.**

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 interfaces of this package to be compiled. To build the new service interface, you should add the following line to it:

In [ ]:

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

- Notes -

In your case, you will also have the message interfaces here:

In [ ]:

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

- End of Notes -

In summarizing, this is the minimum expression of what is needed for the `CMakeList.txt` to work:

In [ ]:

```
cmake_minimum_required(VERSION 3.5)
project(custom_interfaces)

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

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

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

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

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

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

ament_package()
```

4.- Modify package.xml

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_format3.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="user@todo.todo">user</maintainer>
  <license>TODO: License declaration</license>

  <buildtool_depend>ament_cmake</buildtool_depend>

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

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

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

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

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

Execute in Shell #1

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 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 service_messages created through the colcon build.

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

- END -

To verify that your service_message has been created successfully, type into your webshell: `ros2 interface show custom_interfaces/srv/MyCustomServiceMessage`

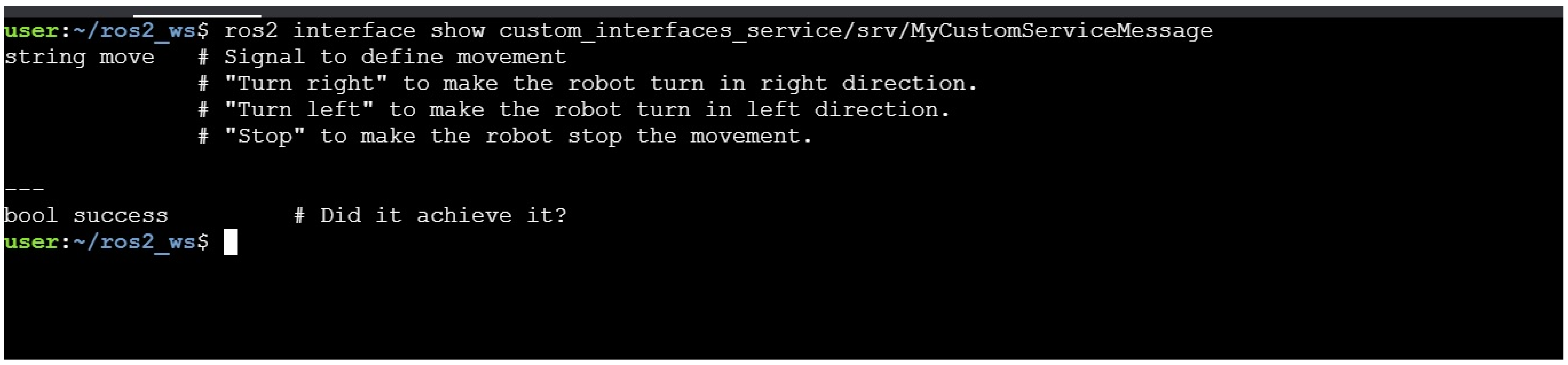
If the structure of the Service type appears, it means that your interface has been created successfully and is ready to be used in your ROS2 programs.

Execute in Shell #1

In [ ]:

```
ros2 interface show custom_interfaces/srv/MyCustomServiceMessage
```

Shell #1 Output



```
user:~/ros2_ws$ ros2 interface show custom_interfaces_service/srv/MyCustomServiceMessage
string move    # Signal to define movement
               # "Turn right" to make the robot turn in right direction.
               # "Turn left" to make the robot turn in left direction.
               # "Stop" to make the robot stop the movement.

---
bool success          # Did it achieve it?
user:~/ros2_ws$
```

- End of Example 4.5 -

4.7 Use a Custom Interface

Use a custom interface to understand how to use the interface you created in Example 4.5 of the previous section.

- Example 4.6 -

1.- Create a new package named **movement_pkg** on your `~/ros2_ws/src` directory.

Execute in Shell #1

In [ ]:

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

In [ ]:

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

You will need to import the `custom_interfaces` package to use the new server type you created.

2.- Create two new files named **movement_server.py** and **movement_client.py** inside the **movement_pkg** folder you created in the previous step.

Inside the file **movement_server.py**, write the following code:

movement_server.py

In [ ]:

```python
# import the Twist module from geometry_msgs messages interface
from geometry_msgs.msg import Twist
# import the MyCustomServiceMessage module from custom_interfaces package
from custom_interfaces.srv import MyCustomServiceMessage
# import the ROS2 Python client libraries
import rclpy
from rclpy.node import Node

class Service(Node):

    def __init__(self):
        # Here you have the class constructor

        # call the class constructor to initialize the node as service_stop
        super().__init__('movement_server')
        # create the Service Server object
        # defines the type, name, and callback function
        self.srv = self.create_service(MyCustomServiceMessage, 'movement', self.custom_service_callback)
        # 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
        self.publisher_ = self.create_publisher(Twist, 'cmd_vel', 10)


    def custom_service_callback(self, request, response):
        # The callback function receives the self-class parameter,
        # received along with two parameters called request and response
        # - receive the data by request
        # - return a result as a response

        # create a Twist message
        msg = Twist()

        if request.move == "Turn Right":
            # define the linear x-axis velocity of /cmd_vel topic parameter to 0.1
            msg.linear.x = 0.1
            # define the angular z-axis velocity of /cmd_vel topic parameter to -0.5 to turn right
            msg.angular.z = -0.5
            # Publish the message to the topic
            self.publisher_.publish(msg)
            # print a pretty message
            self.get_logger().info('Turning to right direction!!')
            # response state
            response.success = True
        elif request.move == "Turn Left":
            # define the linear x-axis velocity of /cmd_vel topic parameter to 0.1
            msg.linear.x = 0.1
            # define the angular z-axis velocity of /cmd_vel topic parameter to 0.5 to turn left
            msg.angular.z = 0.5
            # Publish the message to the topic
            self.publisher_.publish(msg)
            # print a pretty message
            self.get_logger().info('Turning to left direction!!')
            # response state
            response.success = True
        elif request.move == "Stop":
            # define the linear x-axis velocity of /cmd_vel topic parameter to 0
            msg.linear.x = 0.0
            # define the angular z-axis velocity of /cmd_vel topic parameter to 0
            msg.angular.z = 0.0
            # Publish the message to the topic
            self.publisher_.publish(msg)
            # print a pretty message
            self.get_logger().info('Stop there!!')
            # response state
            response.success = True
        else:
            # response state
            response.success = False

        # return the response parameter
        return response


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


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

Inside the file **movement_client.py** write the following code:

movement_client.py

In [ ]:

```
# import the MyCustomServiceMessage module from custom_interfaces package
from custom_interfaces.srv import MyCustomServiceMessage
# import the ROS2 Python client libraries
import rclpy
from rclpy.node import Node
import sys


class ClientAsync(Node):

    def __init__(self):
        # Here you have the class constructor

        # call the class constructor to initialize the node as movement_client
        super().__init__('movement_client')
        # create the Service Client object
        # defines the name and type of the Service Server you will work with.
        self.client = self.create_client(MyCustomServiceMessage, 'movement')
        # checks once per second if a Service matching the type and name of the client is available.
        while not self.client.wait_for_service(timeout_sec=1.0):
            # if it is not available, a message is displayed
            self.get_logger().info('service not available, waiting again...')

        # create an Empty request
        self.req = MyCustomServiceMessage.Request()


    def send_request(self):

        # send the request
        self.req.move = sys.argv[1]
        # uses sys.argv to access command line input arguments for the request.
        self.future = self.client.call_async(self.req)
        # to print in the console


def main(args=None):
    # initialize the ROS communication
    rclpy.init(args=args)
    # declare the node constructor
    client = ClientAsync()
    # run the send_request() method
    client.send_request()

    while rclpy.ok():
        # pause the program execution, waits for a request to kill the node (ctrl+c)
        rclpy.spin_once(client)
        if client.future.done():
            try:
                # checks the future for a response from the Service
                # while the system is running.
                # if the Service has sent a response, the result will be written
                # to a log message.
                response = client.future.result()
            except Exception as e:
                # Display the message on the console
                client.get_logger().info(
                    'Service call failed %r' % (e,))
            else:
                # Display the message on the console
                client.get_logger().info(
                    'Response state %r' % (response.success,))
            break

    client.destroy_node()
    # shutdown the ROS communication
    rclpy.shutdown()


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

3.- Create a launch file named **movement_server_launch_file.launch.py** to launch the Service Server node you just created.

Execute in Shell #1

In [ ]:

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

In [ ]:

```
mkdir launch
```

In [ ]:

```
cd launch
```

In [ ]:

```
touch movement_server_launch_file.launch.py
```

In [ ]:

```
chmod +x movement_server_launch_file.launch.py
```

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

movement_server_launch_file.launch.py

In [ ]:

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

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

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

setup.py

In [ ]:

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

package_name = 'movement_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': [
        'movement_server = movement_pkg.movement_server:main',
        'movement_client = movement_pkg.movement_client:main'
        ],
    },
)
```

5.- Compile your package as you know how to.

Execute in Shell #1

In [ ]:

```
cd ~/ros2_ws
```

In [ ]:

```
colcon build --packages-select movement_pkg
```

In [ ]:

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

7.- Launch the Service Server node on your shell.

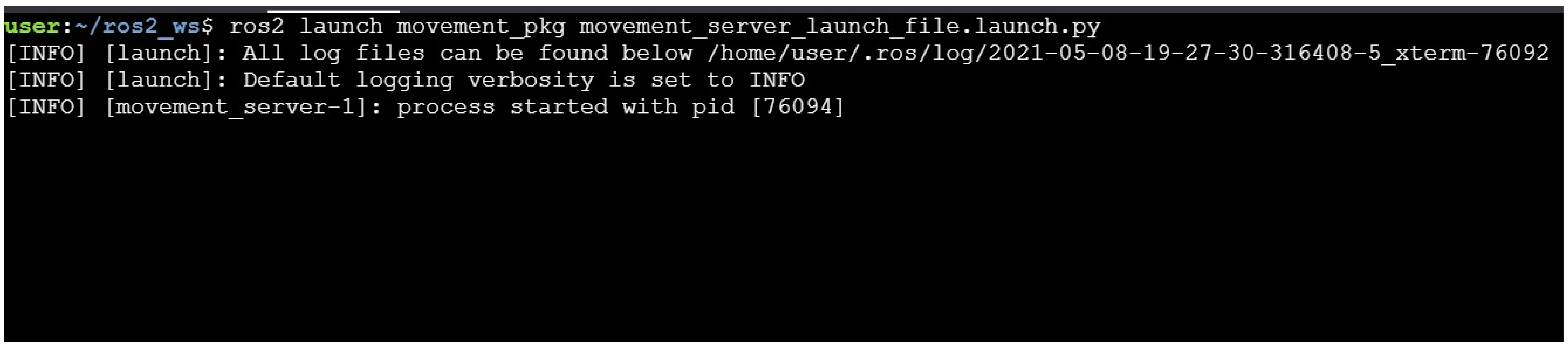
Execute in Shell #1

In [ ]:

```
ros2 launch movement_pkg movement_server_launch_file.launch.py
```

You should get a result similar to the following.

Shell #1 Output



```
user:~/ros2_ws$ ros2 launch movement_pkg movement_server_launch_file.launch.py
[INFO] [launch]: All log files can be found below /home/user/.ros/log/2021-05-08-19-27-30-316408-5_xterm-76092
[INFO] [launch]: Default logging verbosity is set to INFO
[INFO] [movement_server-1]: process started with pid [76094]
```

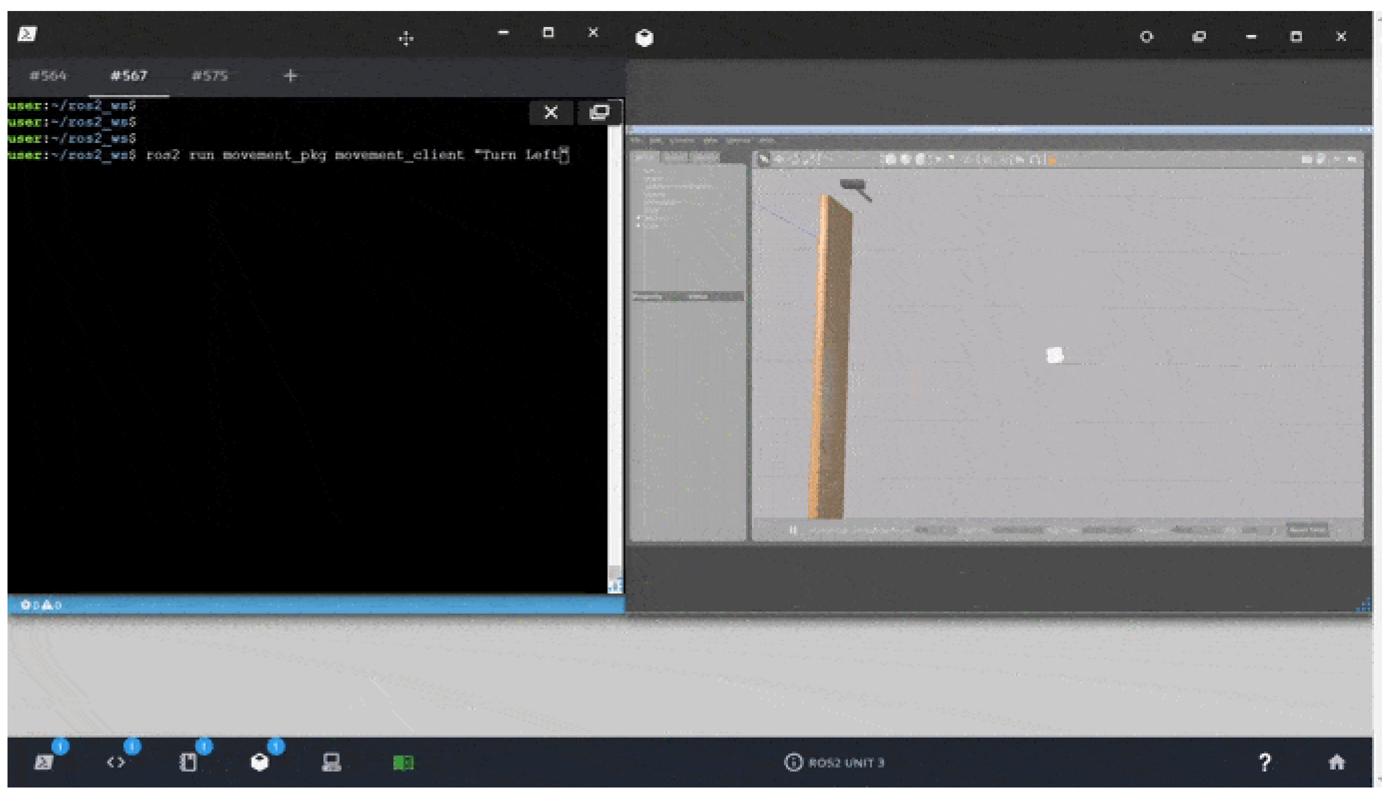
8.- Turn the robot to the left, executing the following command in your Shell #2

Execute in Shell #1

In [ ]:

```
ros2 run movement_pkg movement_client "Turn Left"
```

Shell #1 Output



```
user:~/ros2_ws$
user:~/ros2_ws$
user:~/ros2_ws$
user:~/ros2_ws$ ros2 run movement_pkg movement_client "Turn Left"
```

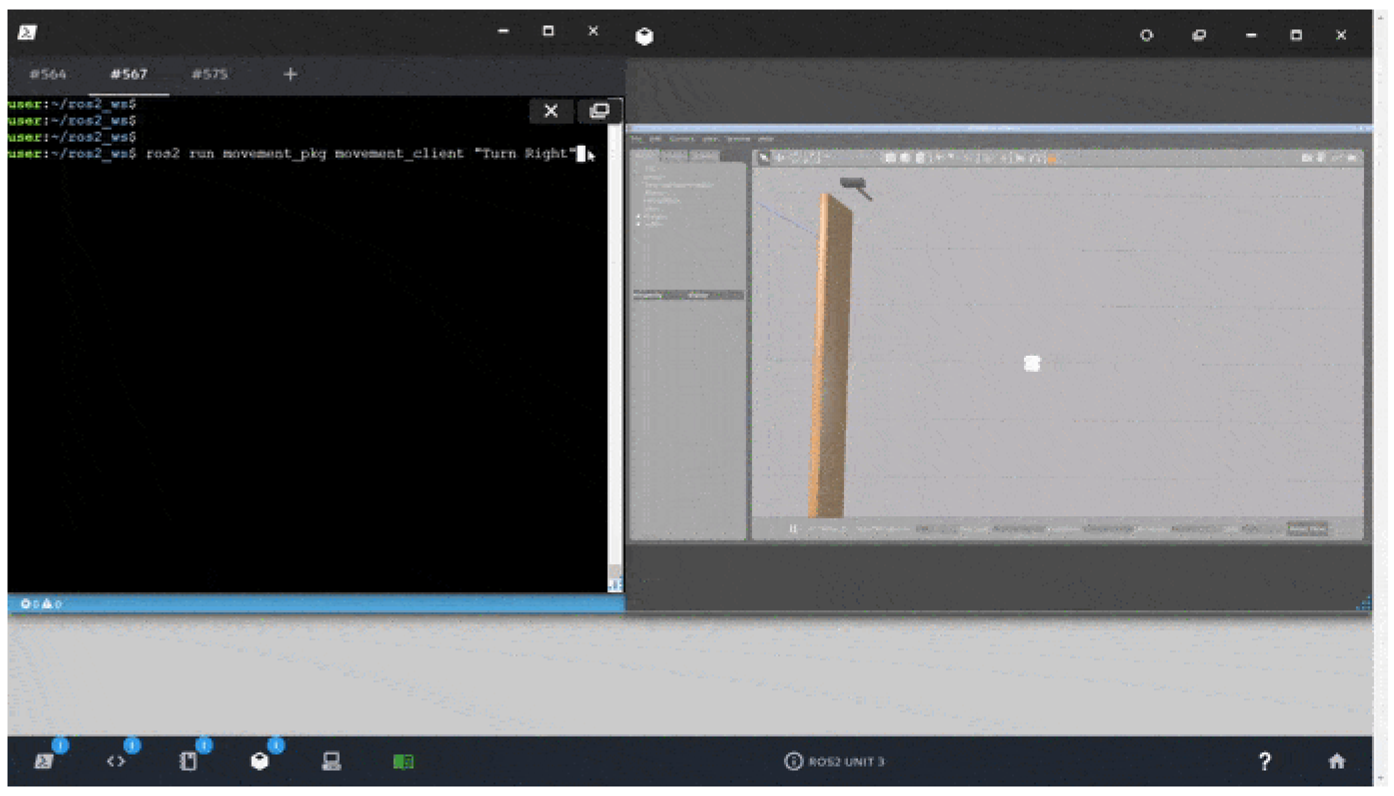
9.- Turn the robot to the right, executing the following command in your Shell #2

Execute in Shell #1

In [ ]:

```
ros2 run movement_pkg movement_client "Turn Right"
```

Shell #1 Output



```
user:~/ros2_ws$
user:~/ros2_ws$
user:~/ros2_ws$
user:~/ros2_ws$ ros2 run movement_pkg movement_client "Turn Right"
```

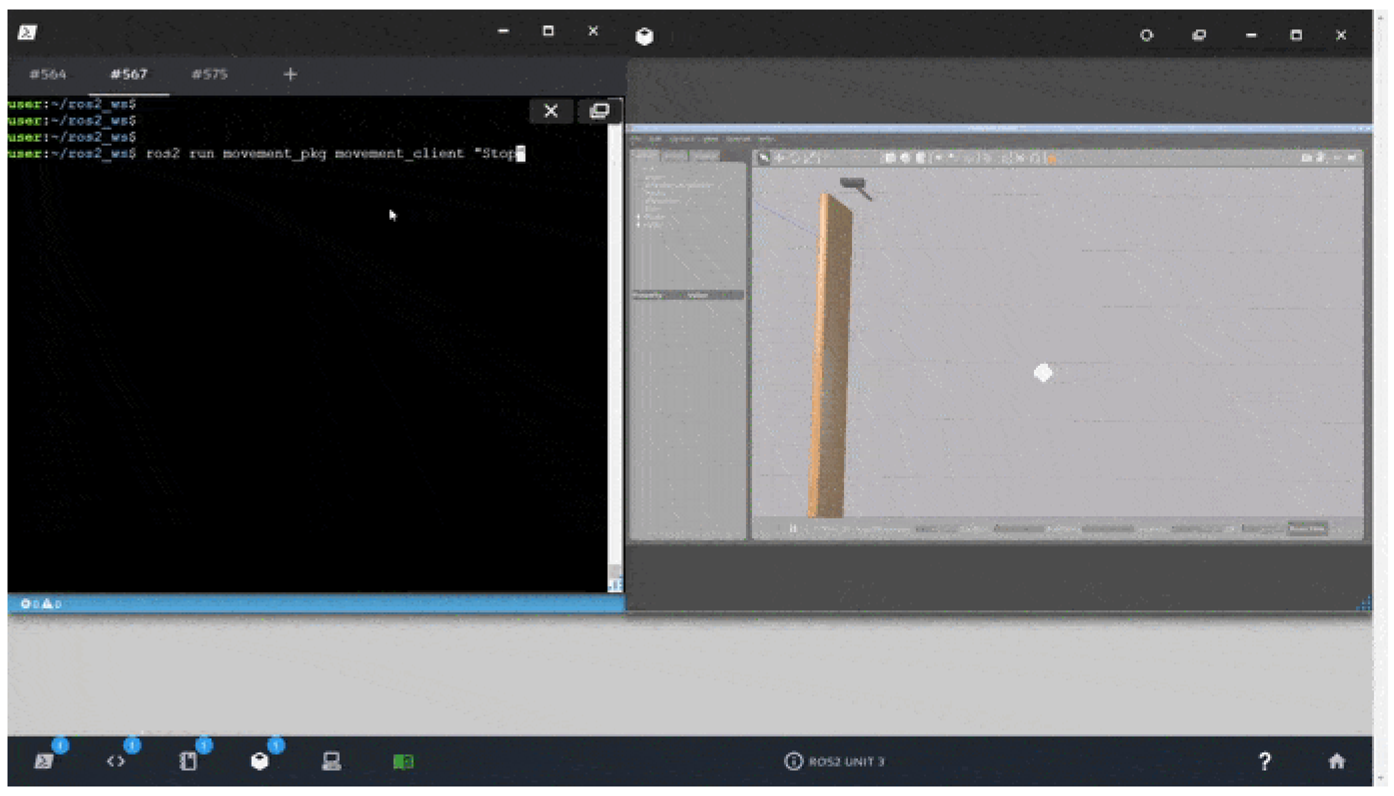
8.- Stop the robot from executing the following command in your Shell #2

Execute in Shell #1

In [ ]:

```
ros2 run movement_pkg movement_client "Stop"
```

Shell #1 Output



```
user:~/ros2_ws$
user:~/ros2_ws$
user:~/ros2_ws$
user:~/ros2_ws$ ros2 run movement_pkg movement_client "Stop
```

- Notes -

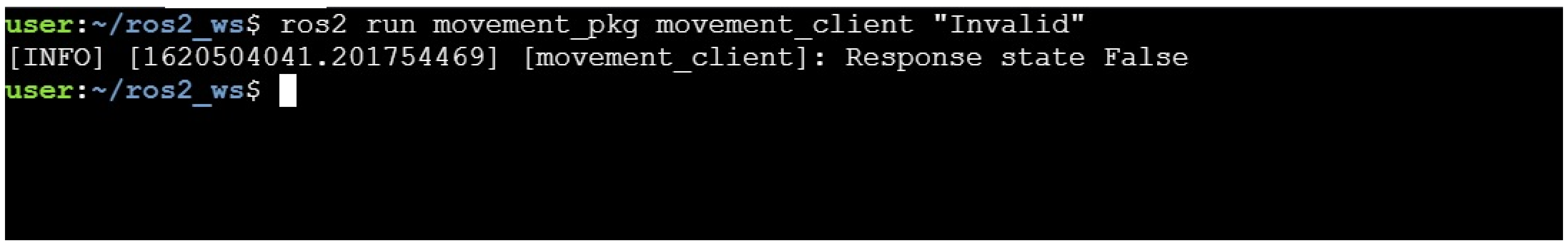
Have you ever wondered what happens when you send an invalid parameter as a request?

Execute in Shell #1

In [ ]:

```
ros2 run movement_pkg movement_client "Invalid"
```

Shell #1 Output



```
user:~/ros2_ws$ ros2 run movement_pkg movement_client "Invalid"
[INFO] [1620504041.201754469] [movement_client]: Response state False
user:~/ros2_ws$
```

As you can see, because of how you created the custom interface and the Service Server, you will assign a negative bool as a response if the parameter is invalid.

4.7.1 Code Review

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

Start with the server code! But, first, pay attention to the Service creation and how you use the new Service type you created in the previous section:

In [ ]:

```
self.srv = self.create_service(
            MyCustomServiceMessage, 'movement', self.custom_service_callback)
```

On the Service callback, you will decide how to move the robot depending on the request received:

In [ ]:

```
if request.move == "Turn Right":

    ...

elif request.move == "Turn Left":

    ...

elif request.move == "Stop":

    ...

else:

    ...
```

As you can see, you are checking the `request.move` value. This refers to the `move` variable of the request part of the Service type:

In [ ]:

```
string move
---
bool success
```

Therefore, to access the response part of the Service type, you will have to use `request.success` :

In [ ]:

```
response.success = True
```

Have a look at the Client code!

The only "special" code used here is the following:

In [ ]:

```
self.req.move = sys.argv[1]
```

This will capture the first argument used in the command and store it in `self.req.move` , which as you already know, is the request part of the Service type.

So, if you use a command like the below one:

In [ ]:

```
ros2 run movement_pkg movement_client "Stop"
```

Here, `"Stop"` is the argument, and it will be stored as a string in `self.req.move` .

- End of Example 4.6 -

4.8 Services Quiz

For evaluation, this quiz will ask you to perform different tasks. For each task, very **specific instructions** will be 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 properly grade your quiz and assign a score. If the names you use are different from the ones specified in the exam instructions, your exercise will be marked as **FAILED**, even though it works correctly.

TO DO: Create a Service named **/turn**. This Service receives a custom Service interface to make the robot spin:

- **To a specified direction (`left` or `right`)**
- **With a specified angular velocity (in radians/second)**
- **During a specified time (in seconds).**

This new Service will have to use Service messages of the **Turn** type. The **Turn.srv** file will be something like this:

In [ ]:

```
string direction              # Direction to spin (right or left)
float64 angular_velocity      # Angular Velocity (in rad/s)
int32 time                    # Duration of the spin (in seconds)
---
bool success                  # Did it achieve it?
```

- **Create a new package named services_quiz_srv, where you will add the new Service interface.**

NOTE: This package will ONLY contain the `Turn.srv` custom interface.

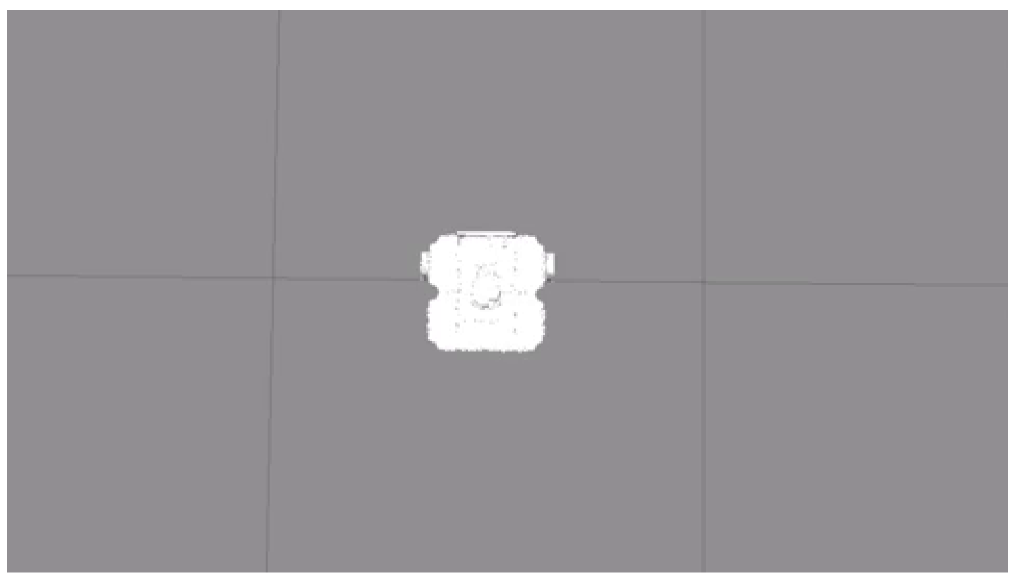
- **Create another package named services_quiz. You will put the Python scripts that contain the Service Server and Client in this package, and also the launch files to start them.**

- **Create the Python script that will contain the code of the Service Server.**

- **Use the data received in the request part of the message to make the robot spin:**
  - **The robot will spin to one side or another depending on the `direction` value.**
  - **Use the `angular_velocity` value to determine the velocity at which the robot will spin.**
  - **Use the `time` value to determine the duration of the spin.**
  - **Finally, it must return True if everything went okay in the success variable.**
- **Create a new launch file, named services_quiz_server.launch.py, that launches the new Service.**
- **Test that when calling the /turn Service, the robot turns accordingly. This means:**
  - **The robot spins in the specified direction.**
  - **The robot spins at the specified angular velocity.**
  - **The robot spins during the specified time.**
- Create another Python script that contains the Service Client code. This Client calls the Service **/turn**, and makes the robot spin with the following specifications:
  - **It will spin to the right.**
  - **It will spin at an angular velocity of 0.2 rad/s.**
  - **It will spin during 10 seconds.**
- Create also the launch that starts the Service Client and name it **services_quiz_client.launch.py**.

- **Test that when running the Client, the robot turns accordingly.**

The resulting motion should be something like this:



Specifications

- **The name of the package where you'll place all the code related to the Service Server and client will be services_quiz.**
- **The name of the package where you'll place all the code related to the Service interface will be services_quiz_srv.**
- **The name of the launch file that will start your Service Server will be `services_quiz_server.launch.py` . This should only start the Service Server, and not the Service Client.**
- **The name of the Service will be /turn.**
- **The name of your Service message file will be Turn.srv.**
- **The name of the launch file that will start your Service Client will be `services_quiz_client.launch.py` . This launch file should not start the Service Server, only the Service Client.**
- **Your Service Client script must exit cleanly after it completes the movements and receives a response from the Server. Please also ensure that it completes the movements within one minute.**
  - **If it does not exit after three minutes, it will be killed automatically, and you may not get credit for work done even if it works as expected.**

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. **Is the custom Service message created successfully?**
4. **Does the Service Server launch file start the Service Server?**
5. **Does the Service Client launch file start the Service Client?**
6. **Did the robot perform the expected movements?**

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 score. For that, click on the following button at the top of this notebook.

✓

Final Mark

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

✓ ✗

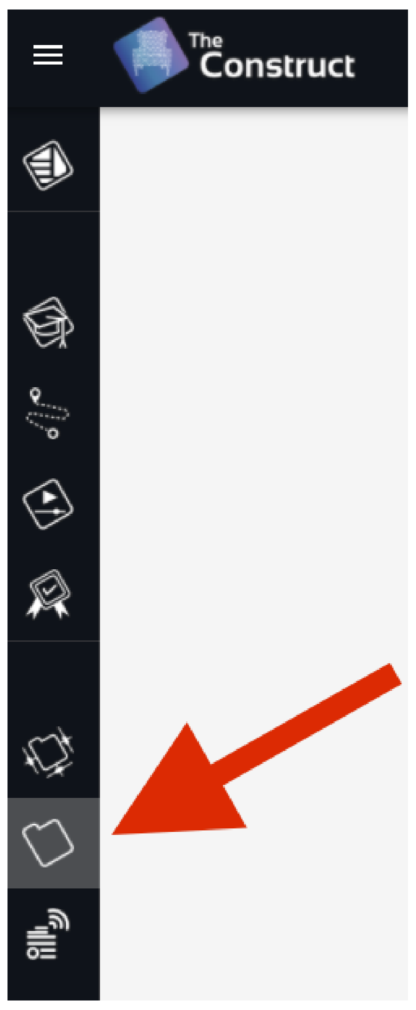
It is now time that you start the project for this course!

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.

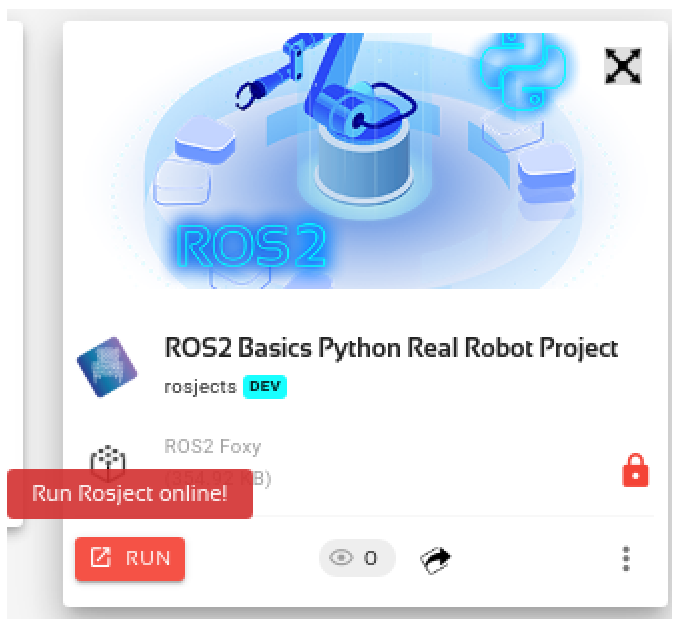
The ROSDS is included with your subscription and is integrated inside The Construct. So no extra effort needs to be made by you. Well, you will need to expend some extra learning effort! But that is why you are here!

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 the PART II 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 II of the rosject before attempting the next unit in this course!

English proofread