ROS2 Basics in 5 Days (Python)

Unit 6 Understanding ROS2 Actions

- Summary -

Estimated time to completion: **4 h**

What will you learn with this unit?

- Understand an Action
- How to call an Action Server
- How to write an Action Server
- How to create your own Action Interface
- How to use your Action Interface

- End of Summary -

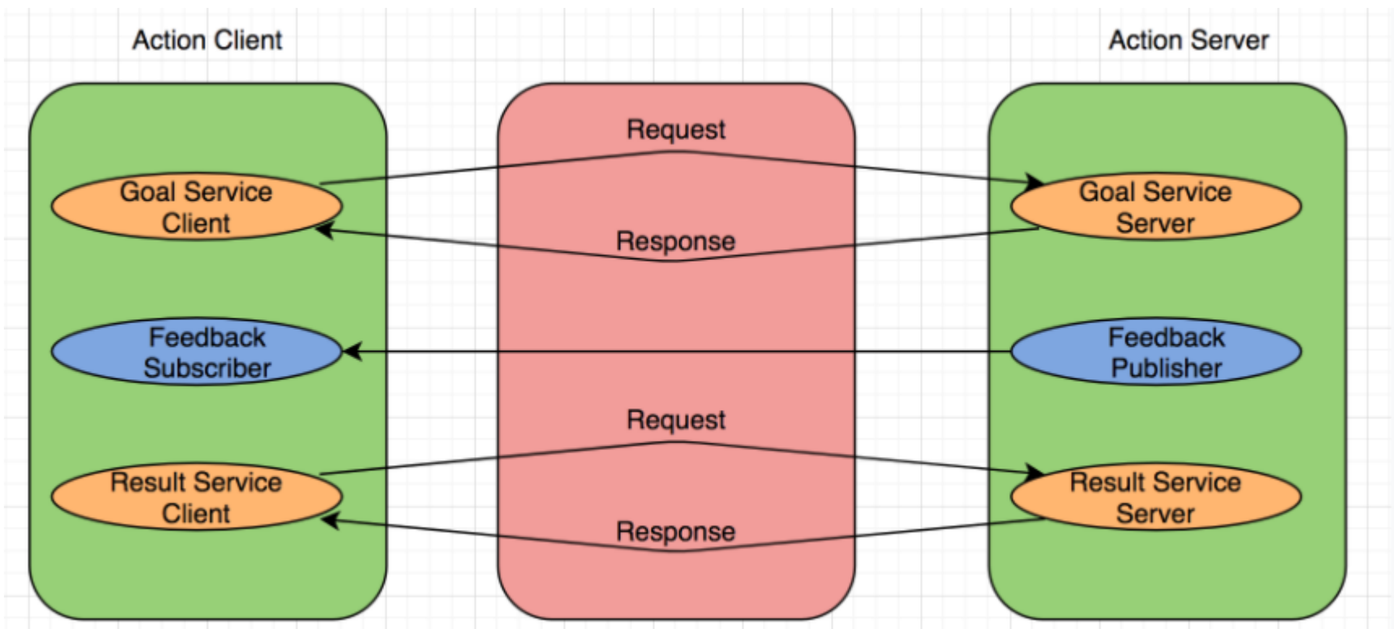
6.1 What is an Action in ROS2?

Actions are very similar to services. When you call an Action, you are calling functionality that another node is providing. Also, Actions are based on a Client-Server model - the same as with Services.

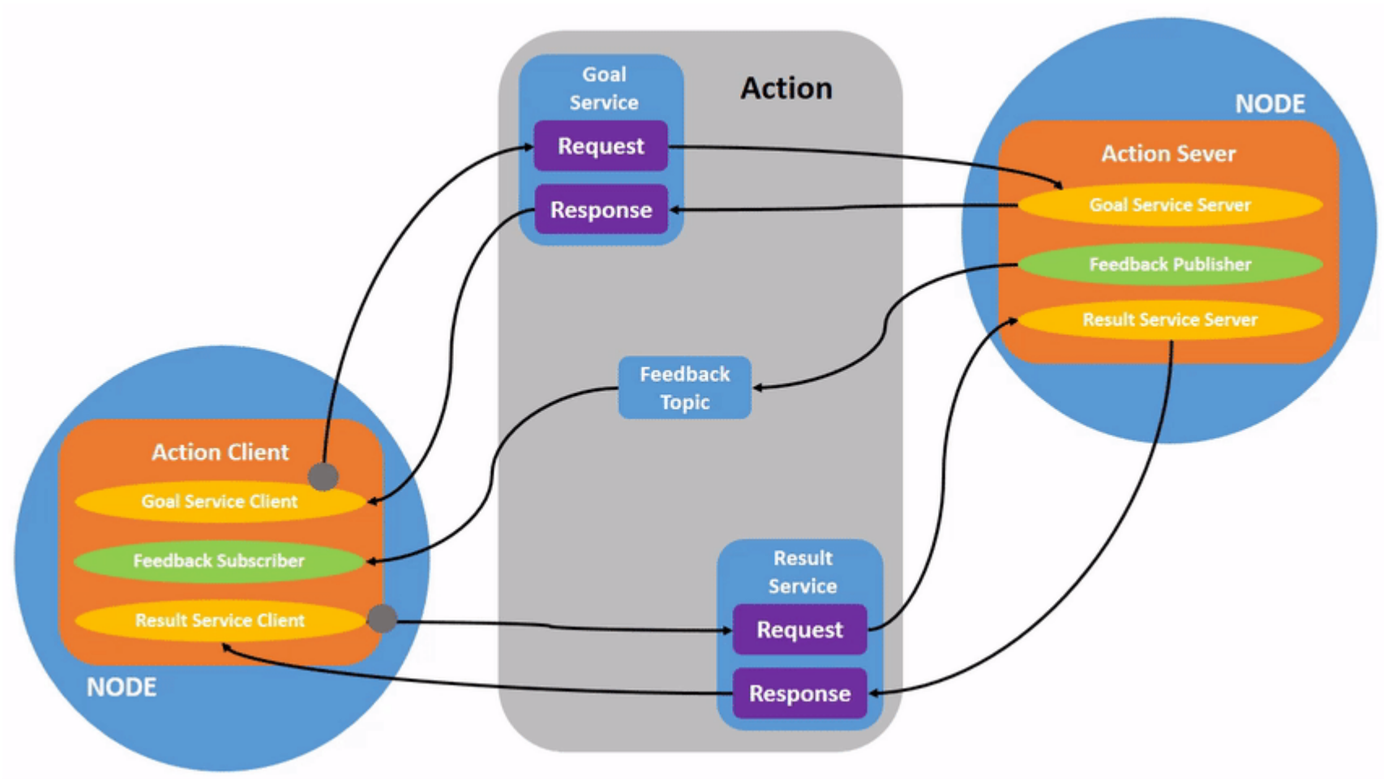
However, there are two main differences between Actions and Services:

- First, **Actions are preemptable**. This means that you can cancel an Action while it is being executed.
- Second, **Actions provide feedback**. This means that, while the Action is being executed, the Server can send feedback back to the Client.

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



As you can see from the graph above, Actions use Services for handling the goal and the result and use Topics to handle the feedback. You can visualize the entire workflow in the following animated image:



The animated image is taken from the official ROS2 documentation.

Do not worry if you do not fully understand it right now, since it contains many concepts. Finally, keep the two following points in mind:

- The node that provides the Action functionality has to contain an **Action Server**. The Action Server allows other nodes to call that Action functionality.
- The node that calls to the Action functionality has to contain an **Action Client**. The Action Client allows a node to connect to the Action Server of another node.

Now, see an Action in action. (I am so funny!)

In summarizing, the workflow goes like this:

1. The **Client sends a goal** to the Server. This will trigger the "start" of the Action.
2. The **Server sends feedback** to the Client while the Action is taking place.
3. Once the Action finishes, the **Server returns a response** to the Client.

Start with some practice to better understand these ideas!

- Example 6.1 -

Execute in Shell #1

In [ ]:

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

In [ ]:

```
ros2 launch turtlebot3_as action_server.launch.py
```

To find which actions are available on a robot, use the command ros2 action list.

Execute in Shell #2

In [ ]:

```
ros2 action list
```

One of the actions in the list should be the `/turtlebot3_as`:

Shell #2 Output

```
...
...
/turtlebot3_as
...
...
```

You can also get data from a specific Action with the following command:

Execute in Shell #2

In [ ]:

```
ros2 action info /turtlebot3_as
```

Shell #2 Output

```
Action: /turtlebot3_as
Action clients: 0
Action servers: 1
    /t3_action_server
```

Also, if you add the suffix `-t` to the command above, you will get data from the action interface used:

Execute in Shell #2

In [ ]:

```
ros2 action info /turtlebot3_as -t
```

Shell #2 Output

```
Action: /turtlebot3_as
Action clients: 0
Action servers: 1
    /t3_action_server [t3_action_msg/action/Move]
```

This output, which may seem somewhat strange at first reading, tells us that the `/turtlebot3_as` Action uses the interface `t3_action_msg/action/Move`. It is there, at the end of the last line. In more general terms, it has the form:

In [ ]:

```
<pkg_name>/action/<interface_name>
```

This means that the Action uses an interface named **Move** defined inside a package named **t3_action_msg**. In a different Action Server, the name of the package and the interface may be different. All Action Interfaces are defined inside a folder called **action**.

You can also get more data about this interface with the following command:

Execute in Shell #2

In [ ]:

```
ros2 interface show t3_action_msg/action/Move
```

Shell #2 Output

```
int32 secs
---
string status
---
string feedback
```

All right! Now that you have gathered some data about the Action Server, call it!

To call an Action Server, you can use the command **ros2 action send_goal**. The structure of the command is the following:

In [ ]:

```
ros2 action send_goal <action_name> <action_type> <values>
```

Now that you have all the data about the Action Server, you can complete the command:

Execute in Shell #2

In [ ]:

```
ros2 action send_goal /turtlebot3_as t3_action_msg/action/Move "{secs: 5}"
```

Shell #2 Output

```
Waiting for an Action Server to become available...
Sending goal:
     secs: 5

Goal accepted with ID: fd252aaa5fee48d595870f0b2a1c9705

Result:
    status: Finished action server. Robot moved during 5 seconds

Goal finished with status: SUCCEEDED
```

Shell #1 Output

```
[turtlebot3_as-1] [INFO] [1616759072.854181251] [t3_action_server]: Received goal request with secs 5
[turtlebot3_as-1] [INFO] [1616759072.855170407] [t3_action_server]: Executing goal
[turtlebot3_as-1] [INFO] [1616759072.855293596] [t3_action_server]: Publish feedback
[turtlebot3_as-1] [INFO] [1616759073.855328070] [t3_action_server]: Publish feedback
[turtlebot3_as-1] [INFO] [1616759074.857161358] [t3_action_server]: Publish feedback
[turtlebot3_as-1] [INFO] [1616759075.855388797] [t3_action_server]: Publish feedback
[turtlebot3_as-1] [INFO] [1616759076.857644650] [t3_action_server]: Publish feedback
[turtlebot3_as-1] [INFO] [1616759077.855392256] [t3_action_server]: Goal succeeded
```

But wait! You told me that Actions provide feedback, and I can see in the Server output that some kind of feedback is being published. So, where is the feedback?

You are right. Actions provide feedback. And you can indeed visualize this feedback. However, you need to specify that you want to visualize the feedback when you call the Action Server.

Try the following command:

Execute in Shell #2

In [ ]:

```
ros2 action send_goal -f /turtlebot3_as t3_action_msg/action/Move "{secs: 5}"
```

- Notes -

Note the `-f` argument added to the command, which is the short flag form for `--feedback`. You can use either as you wish.

- End of Notes -

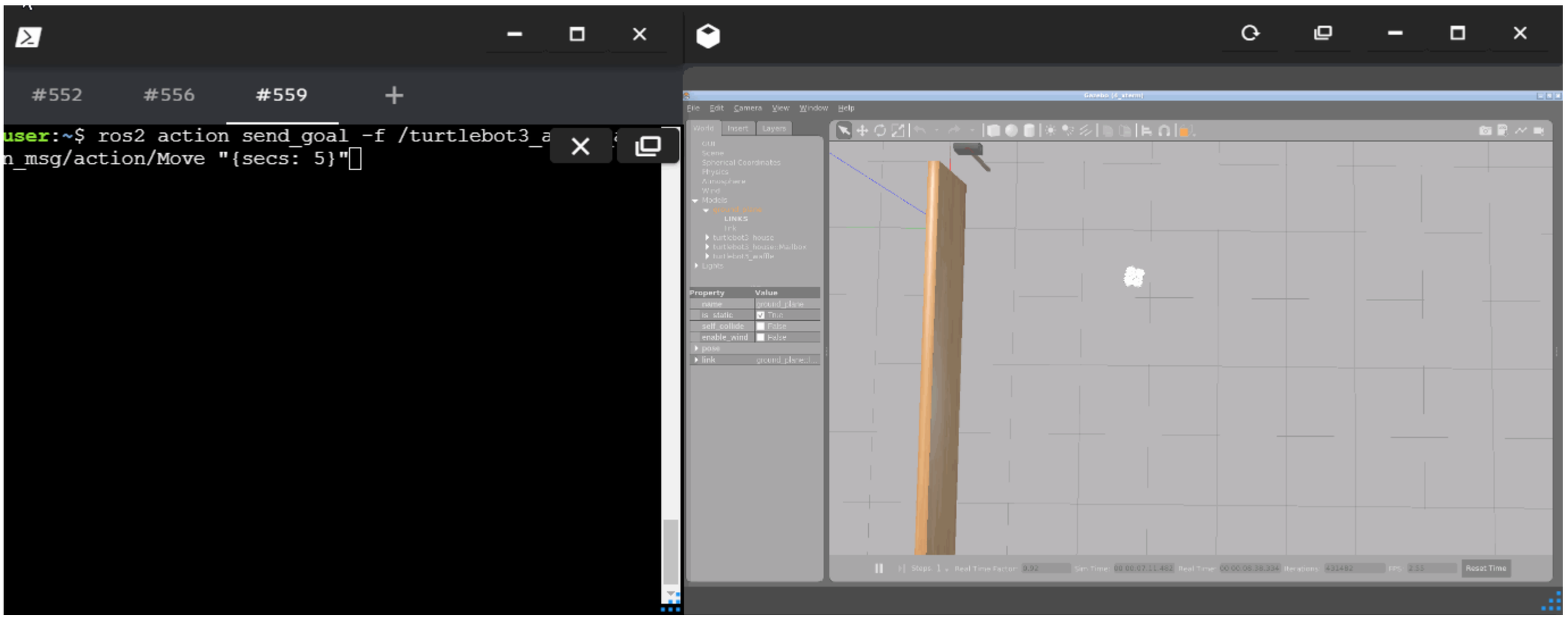
Shell #2 Output

```
Waiting for an Action Server to become available...
Sending goal:
     secs: 5

Feedback:
    feedback: Movint to the left left left...

Goal accepted with ID: 5722a8d8352c4f94b888372ba0a367c4

Feedback:
    feedback: Movint to the left left left...

Feedback:
    feedback: Movint to the left left left...

Feedback:
    feedback: Movint to the left left left...

Feedback:
    feedback: Movint to the left left left...

Result:
    status: Finished action server. Robot moved during 5 seconds

Goal finished with status: SUCCEEDED
```

- End of Example 6.1 -

- Expected Behavior for Example 6.1 -

The robot turns left for five seconds:



IMPORTANT: Before starting the next exercise, stop the previously launched service server by pressing Ctrl + C on the console where you started it (WebShell #1).

- End of Expected Behavior for Example 6.1 -

6.2 Calling an Action Server

The **turtlebot3_as** Action Server is an Action that you can call. If you call it, it will start moving the TurtleBot3 robot forward for the number of seconds specified in the calling message (it is a parameter that you specify in the call).

Calling an Action Server means sending a goal to it. Like Topics and Services, it all works by passing messages around.

- The message of a Topic is composed of a single part: the information the Topic provides.
- The message of a Service has two parts: the request and the response.
- **The message of an Action Server is divided into three parts: the goal, the result, and the feedback.**

All of the Action messages used are defined in the **Action** directory of their package.

If you check the `t3_action_msg` package, you will see that it contains a directory called Action. Inside that Action directory, there is a file called `Move.action`. That is the file specifying the type of message the Action uses.

- Notes -

In this case, the `t3_action_msg` package is installed in the system. You can check it out by running the following command:

Execute in Shell

In [ ]:

```
cd /home/simulations/ros2_sims_ws/src/t3_foxy/t3_action_msg/
```

- End of Notes -

Type the following command in a shell to see the message structure:

Execute in Shell #2

In [ ]:

```
ros2 interface show t3_action_msg/action/Move
```

Shell #2 Output

```
int32 secs
---
string status
---
string feedback
```

You can see in the output that the message is composed of three parts:

- **goal**: Consists of a variable called **secs** of type **int32**.
- **result**: Consists of a variable called **status**, which is of type **string**.
- **feedback**: Consists of a variable called **feedback** of type **string**.

```
# goal
int32 secs # the number of seconds the robot will move forward
---
# result
string status # a string indicating the final status when the Action ends
---
# feedback
string feedback # a string that indicates the current status of the robot
```

In the second part of this chapter, you will learn how to create your own action interfaces. However, for now, you must only understand that every time you call an Action, the message implied contains three parts and that each part can contain **more than one** variable.

6.2.1 Actions provide feedback

Because calling an Action Server does not interrupt your thread, Action Servers provide a message called **feedback**. The feedback is a message that the Action Server occasionally generates to indicate how the Action is going (informing the caller of the status of the requested Action). It is generated while the Action is in progress.

6.2.2 How to call an Action Server

The way you call an Action Server is by implementing an **Action Client**.

In Example 6.1, you created an **action_client** using the command-line tools, with the command **ros2 action send_goal**. This is very useful for testing or debugging actions. However, calling an Action Server using the command-line tools has some limitations. For instance, it does not allow you to customize the Client.

Instead, you will usually have to implement an Action Client by creating a program.

The following is a self-explanatory example of how to implement an Action Client that calls the **turtlebot3_as** Action Server and makes it move forward for five seconds.

- Example 6.2 -

First, create a new package where you will place your Action Client code.

Execute in Shell #1

In [ ]:

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

In [ ]:

```
ros2 pkg create my_action_client --build-type ament_python --dependencies rclpy rclpy.action t3_action_msg
```

Now, inside the **my_action_client** folder, create a new **Python** file named **action_client.py**. Then, you can paste the code below into the script.

action_client.py

In [ ]:

```
import rclpy
from rclpy.action import ActionClient
from rclpy.node import Node
from t3_action_msg.action import Move


class MyActionClient(Node):

    def __init__(self):
        super().__init__('my_action_client')
        self._action_client = ActionClient(self, Move, 'turtlebot3_as')

    def send_goal(self, seconds):
        goal_msg = Move.Goal()
        goal_msg.secs = seconds

        self._action_client.wait_for_server()
        self._send_goal_future = self._action_client.send_goal_async(
            goal_msg, feedback_callback=self.feedback_callback)

        self._send_goal_future.add_done_callback(self.goal_response_callback)

    def goal_response_callback(self, future):
        goal_handle = future.result()
        if not goal_handle.accepted:
            self.get_logger().info('Goal rejected :(')
            return

        self.get_logger().info('Goal accepted :)')

        self._get_result_future = goal_handle.get_result_async()
        self._get_result_future.add_done_callback(self.get_result_callback)

    def get_result_callback(self, future):
        result = future.result().result
        self.get_logger().info('Result: {0}'.format(result.status))
        rclpy.shutdown()

    def feedback_callback(self, feedback_msg):
        feedback = feedback_msg.feedback
        self.get_logger().info(
            'Received feedback: {0}'.format(feedback.feedback))


def main(args=None):
    rclpy.init(args=args)

    action_client = MyActionClient()

    action_client.send_goal(5)

    rclpy.spin(action_client)


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

Modify the **setup.py**:

setup.py

In [ ]:

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

package_name = 'my_action_client'

setup(
    name='my_action_client',
    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': [
            'example62 = my_action_client.action_client:main'
        ],
    },
)
```

Create a launch file named **example62_launch_file.launch.py**:

Execute in Shell #1

In [ ]:

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

In [ ]:

```
mkdir launch
```

In [ ]:

```
cd ~/ros2_ws/src/my_action_client/launch
```

In [ ]:

```
touch example62_launch_file.launch.py
```

In [ ]:

```
chmod +x example62_launch_file.launch.py
```

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

example62_launch_file.launch.py

In [ ]:

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

def generate_launch_description():
    return LaunchDescription([
        Node(
            package='my_action_client',
            executable='example62',
            output='screen'),
    ])
```

Compile your package:

Execute in Shell #1

In [ ]:

```
cd ~/ros2_ws
```

In [ ]:

```
colcon build --packages-select my_action_client
```

In [ ]:

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

Then, launch the Action Server node from Example 6.1 on your Shell #1.

Execute in Shell #1

In [ ]:

```
ros2 launch turtlebot3_as action_server.launch.py
```

Finally, launch the Action Client node on your Shell #2.

Execute in Shell #2

In [ ]:

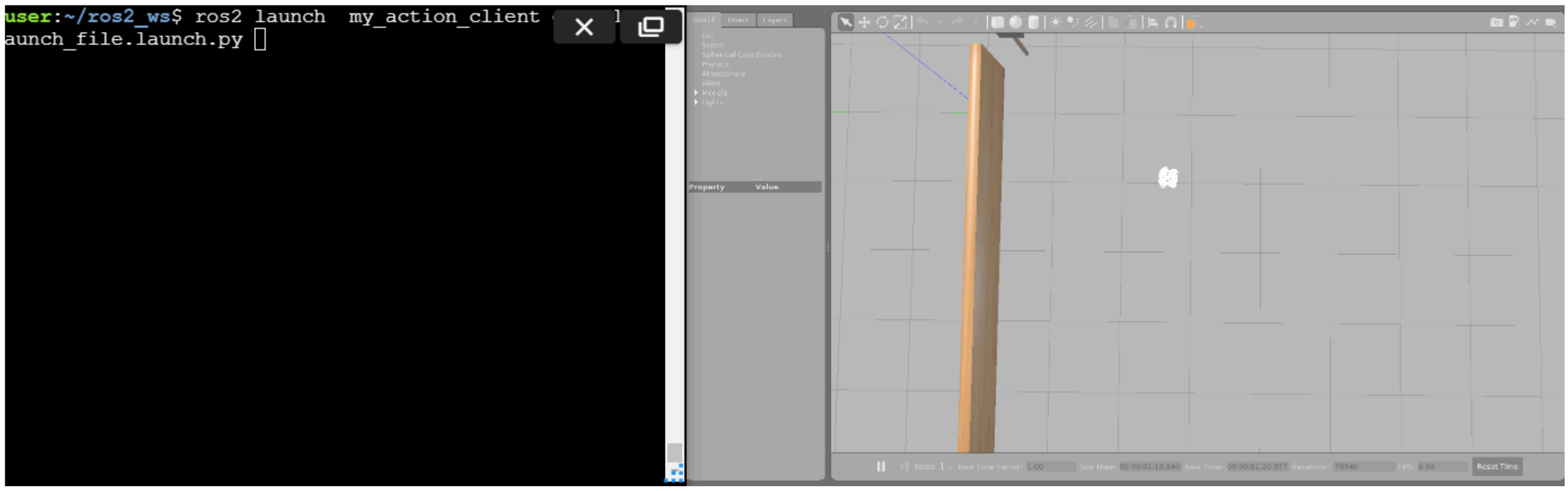
```
source ~/ros2_ws/install/setup.bash
```

In [ ]:

```
ros2 launch my_action_client example62_launch_file.launch.py
```

- Expected Behavior for Example 6.2 -

The robot turns left for five seconds:



You should get a result similar to the following:

Shell #2 Output

```
[INFO] [launch]: All log files can be found below /home/user/.ros/log/2021-05-08-18-01-29-328260-4_xterm-27910
[INFO] [launch]: Default logging verbosity is set to INFO
[INFO] [example42-1]: process started with pid [27912]
[example42-1] [INFO] [1620496889.921096796] [my_action_client]: Goal accepted :)
[example42-1] [INFO] [1620496889.923950338] [my_action_client]: Received feedback:Movint to the left left left...
[example42-1] [INFO] [1620496890.901247019] [my_action_client]: Received feedback:Movint to the left left left...
[example42-1] [INFO] [1620496891.895880131] [my_action_client]: Received feedback:Movint to the left left left...
[example42-1] [INFO] [1620496892.895953418] [my_action_client]: Received feedback:Movint to the left left left...
[example42-1] [INFO] [1620496893.896108731] [my_action_client]: Received feedback:Movint to the left left left...
[example42-1] [INFO] [1620496894.896104351] [my_action_client]: Result: Finished action server. Robot moved during 5 se
conds
[INFO] [example42-1]: process has finished cleanly [pid 27912]
```

- End of Expected Behavior for Example 6.2 -

- End of Example 6.2 -

6.2.3 Action Client code explanation

In this section, you will break down the code you developed in Section 6.2.2, where I showed you how to write an Action Client. Begin with the first section, where you will import the necessary libraries.

In [ ]:

```
import rclpy
from rclpy.action import ActionClient
from rclpy.node import Node

from t3_action_msg.action import Move
```

As you may have noticed, in the first three lines, you do nothing more than import the ROS2 Python client libraries to work with Actions (`rclpy.action`) and nodes (`rclpy.node`). You can also see that this is where you will import the interfaces you work with, in this case, (`t3_action_msg.action`).

Now jump to the `main` function.

In [ ]:

```
def main(args=None):
    rclpy.init(args=args)

    action_client = MyActionClient()

    action_client.send_goal(5)

    rclpy.spin(action_client)
```

Here you are creating an instance of the **`MyActionClient()`** class and calling its `send_goal()` method, passing it five seconds as a parameter. (In a moment, you will analyze the contents of this method). Then, finally, spin the node with `rclpy.spin()` so that the callback functions are properly executed.

Now continue analyzing the constructor of the class **`MyActionClient()`**:

In [ ]:

```
def __init__(self):
        super().__init__('my_action_client')
        self._action_client = ActionClient(self, Move, 'turtlebot3_as')
```

In the constructor of the class, initialize a ROS2 node named `my_action_client` . Also, and very important, create an `ActionClient` object to which you pass three arguments:

1. The ROS2 node that contains the Action Client: in this case, **`self`** .
2. The type of the Action: **`Move`** (related to the `t3_action_msg` interface of type Action).
3. The Action name: **`turtlebot3_as`** .

Now continue analyzing the `send_goal()` method:

In [ ]:

```
def send_goal(self, seconds):
        goal_msg = Move.Goal()
        goal_msg.secs = seconds

        self._action_client.wait_for_server()

        self._send_goal_future = self._action_client.send_goal_async(goal_msg, feedback_callback=self.feedback_callback)
        self._send_goal_future.add_done_callback(self.goal_response_callback)
```

Start by creating a **`Goal()`** object of the **`Move`** action type. Then, access the **`secs`** variable of the Action goal and assign it the value of `seconds` (which is five in this example).

In [ ]:

```
goal_msg = Move.Goal()
goal_msg.secs = seconds
```

- Notes -

Remember the structure of the action type **`Move`** that you checked earlier:

In [ ]:

```
int32 secs
---
string status
---
string feedback
```

- End of Notes -

Next, wait for the Action Server to be up and running:

In [ ]:

```
self._action_client.wait_for_server()
```

And send the goal to the Server using the **`send_goal_async`** method:

In [ ]:

```
self._send_goal_future = self._action_client.send_goal_async(goal_msg, feedback_callback=self.feedback_callback)
```

You need to provide two arguments for this method:

- A goal message, in this case, **`goal_msg`**
- A callback function for the feedback, in this case, **`self.feedback_callback`**

This **`send_goal_async()`** method returns a future to a goal handle. **This future goal handle will be completed when the Server has processed the goal. (This means it has been accepted or rejected by the Server)**. So, you must assign a callback method to be triggered when the future is completed (the goal has been accepted or rejected). In this case, this method is **`self.goal_response_callback`**:

In [ ]:

```
self._send_goal_future.add_done_callback(self.goal_response_callback)
```

Now have a look at this **`self.goal_response_callback`** method:

In [ ]:

```
def goal_response_callback(self, future):
    goal_handle = future.result()
    if not goal_handle.accepted:
        self.get_logger().info('Goal rejected :(')
        return

    self.get_logger().info('Goal accepted :)')

    self._get_result_future = goal_handle.get_result_async()
    self._get_result_future.add_done_callback(self.get_result_callback)
```

So, this method will be triggered when the goal has been processed. First, check whether the Server has accepted the goal:

In [ ]:

```
goal_handle = future.result()
        if not goal_handle.accepted:
```

Print a message if it has been rejected. If it has been accepted, ask for the result using the **`get_result_async()`** method:

In [ ]:

```
self._get_result_future = goal_handle.get_result_async()
```

Similar to sending the goal, this method will return a future that will be completed when the result is ready. So, you must also assign a callback method to be triggered when this future is completed (the result is ready). In this case, this method is **`self.get_result_callback`**:

In [ ]:

```
self._get_result_future.add_done_callback(self.get_result_callback)
```

Now, have a look at this **`self.get_result_callback`** method:

In [ ]:

```
def get_result_callback(self, future):
    result = future.result().result
    self.get_logger().info('Result: {0}'.format(result.status))
    rclpy.shutdown()
```

This method is very simple. You get the result (**`future.result().result`**), print, and then shut down the node for a clean exit with `rclpy.shutdown()`.

Finally, you have the feedback callback method:

In [ ]:

```
def feedback_callback(self, feedback_msg):
    feedback = feedback_msg.feedback
    self.get_logger().info(
        'Received feedback: {0}'.format(feedback.feedback))
```

Here you get the **`feedback`** string from the **`feedback_msg`** and print it to the screen.

And that is it! You have reviewed the most important parts of the Action Client node code.

6.3 Writing an Action Server

All right! So you already know how to create an Action Client node. But what about the Action Server?

In the following example, you will create a ROS2 node that implements an Action Server.

- Example 6.3 -

First, create a new package where you will place your Action Server code.

In [ ]:

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

In [ ]:

```
ros2 pkg create my_action_server --build-type ament_python --dependencies rclpy rclpy.action t3_action_msg
```

Now, inside the **my_action_server** folder, create a new Python file named **action_server.py**. You can paste the code below into the script:

action_server.py

In [ ]:

```
import rclpy
from rclpy.action import ActionServer
from rclpy.node import Node

from t3_action_msg.action import Move

from geometry_msgs.msg import Twist
import time
class MyActionServer(Node):

    def __init__(self):
        super().__init__('my_action_server')
        self._action_server = ActionServer(self, Move, 'turtlebot3_as_2',self.execute_callback)
        self.cmd = Twist()
        self.publisher_ = self.create_publisher(Twist, 'cmd_vel', 10)



    def execute_callback(self, goal_handle):

        self.get_logger().info('Executing goal...')

        feedback_msg = Move.Feedback()
        feedback_msg.feedback = "Moving to the left left left..."

        for i in range(1, goal_handle.request.secs):

            self.get_logger().info('Feedback: {0} '.format(feedback_msg.feedback))

            goal_handle.publish_feedback(feedback_msg)
            self.cmd.linear.x = 0.3
            self.cmd.angular.z =0.3

            self.publisher_.publish(self.cmd)
            time.sleep(1)

        goal_handle.succeed()

        self.cmd.linear.x = 0.0
        self.cmd.angular.z = 0.0

        self.publisher_.publish(self.cmd)
        result = Move.Result()
        result.status = "Finished action server. Robot moved during 5 seconds"
        self.get_logger().info('Result: {0}'.format(result.status))
        return result

def main(args=None):
    rclpy.init(args=args)

    my_action_server = MyActionServer()

    rclpy.spin(my_action_server)


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

Modify the **setup.py** file:

setup.py

In [ ]:

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

package_name = 'my_action_server'

setup(
    name='my_action_server',
    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': [
            'example63 = my_action_server.action_server:main'
        ],
    },
)
```

Create a launch file named **example63_launch_file.launch.py**:

Execute in Shell #1

In [ ]:

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

In [ ]:

```
mkdir launch
```

In [ ]:

```
cd ~/ros2_ws/src/my_action_server/launch
```

In [ ]:

```
touch example63_launch_file.launch.py
```

In [ ]:

```
chmod +x example63_launch_file.launch.py
```

Inside it, add the necessary code to launch the executable of the **action_server** script.

example63_launch_file.launch.py

In [ ]:

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

def generate_launch_description():
    return LaunchDescription([
        Node(
            package='my_action_server',
            executable='example63',
            output='screen'),
    ])
```

Compile your package:

Execute in Shell #1

In [ ]:

```
cd ~/ros2_ws
```

In [ ]:

```
colcon build --packages-select my_action_server
```

In [ ]:

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

After compiling, launch the action server node on your Shell #1.

Execute in Shell #1

In [ ]:

```
ros2 launch my_action_server example63_launch_file.launch.py
```

- Notes -

Before running the Client node, make sure to update the name of the Action to `turtlebot3_as_2` :

In [ ]:

```
self._action_client = ActionClient(self, Move, 'turtlebot3_as_2')
```

You will need to build again the `my_action_client` package for the change to be applied.

- End of Notes -

Finally, execute your Client node:

Execute in Shell #2

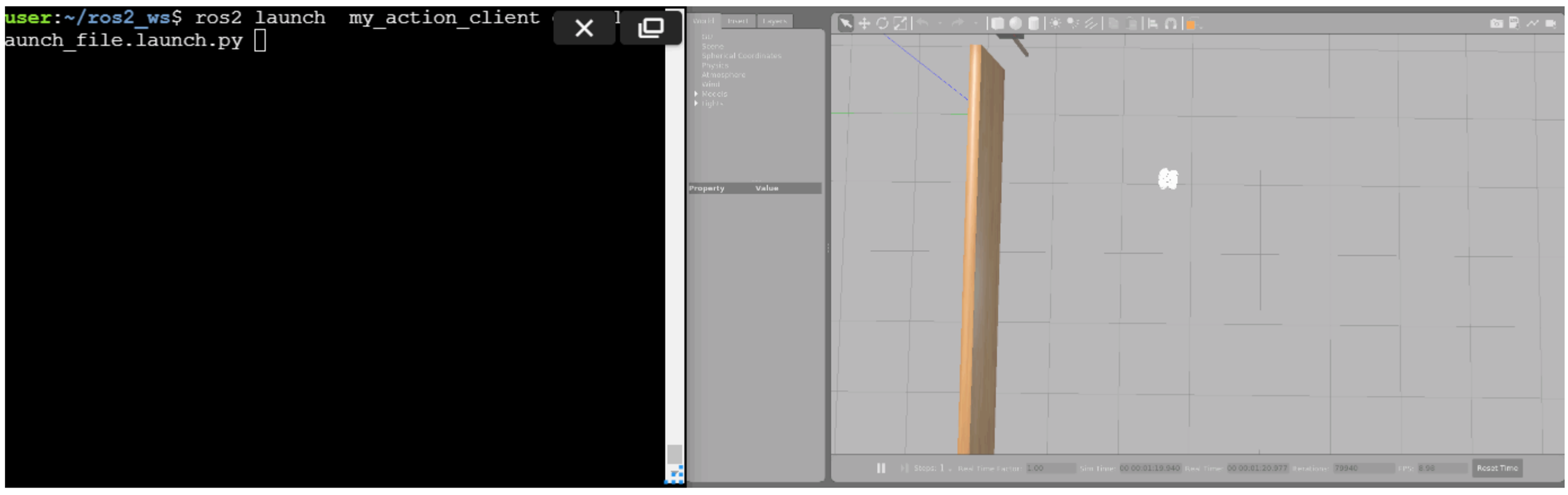
In [ ]:

```
ros2 launch my_action_client example62_launch_file.launch.py
```

- End of Example 6.3 -

- Expected Behavior for Example 6.3 -

The robot turns left for five seconds:



You should get a result similar to the following.

Shell #2 Output

```
[INFO] [launch]: All log files can be found below /home/user/.ros/log/2021-05-08-18-01-29-328260-4_xterm-27910
[INFO] [launch]: Default logging verbosity is set to INFO
[INFO] [example42-1]: process started with pid [27912]
[example42-1] [INFO] [1620496889.921096796] [my_action_client]: Goal accepted :)
[example42-1] [INFO] [1620496889.923950338] [my_action_client]: Received feedback: Moving to the left left left...
[example42-1] [INFO] [1620496890.901247019] [my_action_client]: Received feedback: Moving to the left left left...
[example42-1] [INFO] [1620496891.895880131] [my_action_client]: Received feedback: Moving to the left left left...
[example42-1] [INFO] [1620496892.895953418] [my_action_client]: Received feedback: Moving to the left left left...
[example42-1] [INFO] [1620496893.896108731] [my_action_client]: Received feedback: Moving to the left left left...
[example42-1] [INFO] [1620496894.896104351] [my_action_client]: Result: Finished action server. Robot moved during 5 se
conds
[INFO] [example42-1]: process has finished cleanly [pid 27912]
```

- End of Expected Behavior for Example 6.3 -

6.3.1 Action Server code explanation

Okay. Try to analyze the code in more detail now that you have created your first Action Server.

Begin with the first section, where you will import the necessary libraries.

In [ ]:

```
import rclpy
from rclpy.action import ActionServer
from rclpy.node import Node

from t3_action_msg.action import Move

from geometry_msgs.msg import Twist
import time
```

Pay attention to the import of the `ActionServer` object from `rclpy.action`, which allows you to create an Action Server node. You will also need the interface `Twist` from `geometry_msgs.msg` to send velocity commands to the `/cmd_vel` Topic. Finally, import `time` to use the sleep() function, to count the seconds.

Next, define your class `MyActionServer` that inherits from `Node`:

In [ ]:

```
class MyActionServer(Node):

    def __init__(self):
        super().__init__('my_action_server')
        self._action_server = ActionServer(self, Move, 'turtlebot3_as_2',self.execute_callback)
        self.cmd = Twist()
        self.publisher_ = self.create_publisher(Twist, 'cmd_vel', 10)
```

In the constructor of the class, initialize a ROS2 node named `my_action_server`. Also, and very important, create an `ActionServer` object to which you pass four arguments:

1. The ROS2 node that contains the Action Client: in this case, **`self`**.
2. The type of Action: **`Move`**.
3. The Action name: **`turtlebot3_as`**
4. A callback method to be executed when the Action receives a goal: **`self.execute_callback`**.

Finally, define a Publisher for the Topic **`/cmd_vel`**.

Now continue analyzing the `execute_callback()` method:

In [ ]:

```
def execute_callback(self, goal_handle):
```

This method will be called when the Action Server receives a goal, and it contains the main functionality of the Action (`what the Action will do`).

So start by creating a feedback message (**`Move.Feedback`**) and populating the **`feedback`** string:

In [ ]:

```
feedback_msg = Move.Feedback()
feedback_msg.feedback = "Moving to the left left left..."
```

Next, you have the main logic of your Action:

In [ ]:

```
for i in range(1, goal_handle.request.secs):

    self.get_logger().info('Feedback: '.format(feedback_msg.feedback))

    goal_handle.publish_feedback(feedback_msg)
    self.cmd.linear.x = 0.3
    self.cmd.angular.z =0.3

    self.publisher_.publish(self.cmd)
    time.sleep(1)
```

Here you will iterate over the number of seconds specified in the parameter **`goal_handle.request.secs`**. (In this case, five secs). You do two things here:

- Publish the feedback message: **`goal_handle.publish_feedback(feedback_msg)`**
- Set the desired velocity values and publish this data: **`self.publisher_.publish(self.cmd)`**
- Finally, `sleep` for one second: **`time.sleep(1)`**

When the previous loop finishes, set the goal handle as succeeded and command the robot to stop:

In [ ]:

```
goal_handle.succeed()

self.cmd.linear.x = 0.0
self.cmd.angular.z = 0.0

self.publisher_.publish(self.cmd)
```

Finally, create the result message, fill the **`status`** variable of the result with a final message, and return this result.

In [ ]:

```
result = Move.Result()
result.status = "Finished action server. Robot moved during 5 seconds"
return result
```

6.4 Create an Action Interface

To create your own Action Interface, you must complete the following three steps:

1- Create an `action` directory in your `custom_interfaces` package.

2- Create your `<interface_name>.action` Action Interface file.

- The name of the Action Interface file will later determine the name of the classes to be used in the **Action Server** and/or **Action Client**. ROS2 convention indicates that the name has to be camel-case.
- Remember that the Action Interface file has to contain three parts, each separated by three hyphens.

In [ ]:

```
#goal
message_type goal_var_name
---
#result
message_type result_var_name
---
#feedback
message_type feedback_var_name
```

In this case, it should be like the following:

Move.action

In [ ]:

```
int32 secs
---
string status
---
string feedback
```

If you do not need one part of the message (for example, you do not need to provide feedback), then you can leave that part empty. However, you **must always specify the hyphen separators**.

3- Modify the `CMakeLists.txt` and `package.xml` files to include action interface compilation. Read the detailed description below.

6.4.1 Prepare CMakeLists.txt and package.xml files

You have to edit two files in the package, in the same way as for Topics and Services:

- CMakeLists.txt
- package.xml

Modification of CMakeLists.txt

Call the **rosidl_generate_interfaces** function in your CMakeLists.txt file to create a new Action Interface. To call this function, add the below snippet to your CMakeLists.txt file:

In [ ]:

```
rosidl_generate_interfaces(${PROJECT_NAME}
  "action/Move.action"
)
```

Within the function, specify the name of your Action Interface file.

- Notes -

If you have another interface in this package, you should include all the interfaces. **Do not delete** any interface.

In your case, if you have completed the previous units, you will probably have something like this:

In [ ]:

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

- End of Notes -

To generate Action Interfaces, ensure that you have access to the following packages:

- **rosidl_default_generators**
- **action_msgs**

For this, add the line below to your CMakeLists.txt file:

In [ ]:

```
find_package(action_msgs REQUIRED)
find_package(rosidl_default_generators REQUIRED)
```

As a reference, the **CMakeLists.txt** file of the **t3_action_msg** package, which contains the **Move** Action Interface, looks like this:

Note: `CMakeLists.txt` of `t3_action_msg package`.

CMakeLists.txt

In [ ]:

```
cmake_minimum_required(VERSION 3.5)
project(t3_action_msg)

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

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

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

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

rosidl_generate_interfaces(${PROJECT_NAME}
  "action/Move.action"
)

if(BUILD_TESTING)
  find_package(ament_lint_auto REQUIRED)
  # the following line skips the linter, which checks for copyrights
  # uncomment the line when 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()

ament_package()
```

Modification of package.xml

In the package.xml file, ensure that you have dependencies for the following packages:

- **action_msgs**
- **rosidl_default_generators**

In [ ]:

```
<depend>action_msgs</depend>
<depend>rosidl_default_generators</depend>
```

Also, specify the **rosidl_interface_packages** package as member_of_group:

In [ ]:

```
<member_of_group>rosidl_interface_packages</member_of_group>
```

As a reference, the CMakeLists.txt file of the **t3_action_msg package**, which contains the **Move** action interface, looks like this:

Note: `package.xml` of `t3_action_msg package`

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>t3_action_msg</name>
  <version>0.0.0</version>
  <description>TODO: Package description</description>
  <maintainer email="user@todo.todo">user</maintainer>
  <license>TODO: License declaration</license>

  <buildtool_depend>ament_cmake</buildtool_depend>

  <depend>rclcpp</depend>
  <depend>action_msgs</depend>
  <depend>rosidl_default_generators</depend>

  <member_of_group>rosidl_interface_packages</member_of_group>

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

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

Finally, when everything is correctly set up, compile it:

Execute in Shell #1

In [ ]:

```
cd ~/ros2_ws
```

In [ ]:

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

In [ ]:

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

To verify that your Action Interface has been created correctly, use the following command:

Execute in Shell #1

In [ ]:

```
ros2 interface show custom_interfaces/action/Move
```

You should get something like this:

```
user:~$ ros2 interface show custom_interfaces/action/Move
int32 secs
---
string status
---
string feedback
```

6.4.2 Use this interface in a Action Server and Action Client

This will be easy to test. You will modify your Action Server and Action Client programs created before. This time, instead of importing the action message from `t3_action_msg`, import it from your `custom_interfaces` package.

You could do it like this:

In [ ]:

```
#from t3_action_msg.action import Move
from custom_interfaces.action import Move
```

Once you change these parts of the codes, compile with **colcon build** and launch both codes to see if it is working like it was working before.

Execute in Shell #1

In [ ]:

```
cd ~/ros2_ws
```

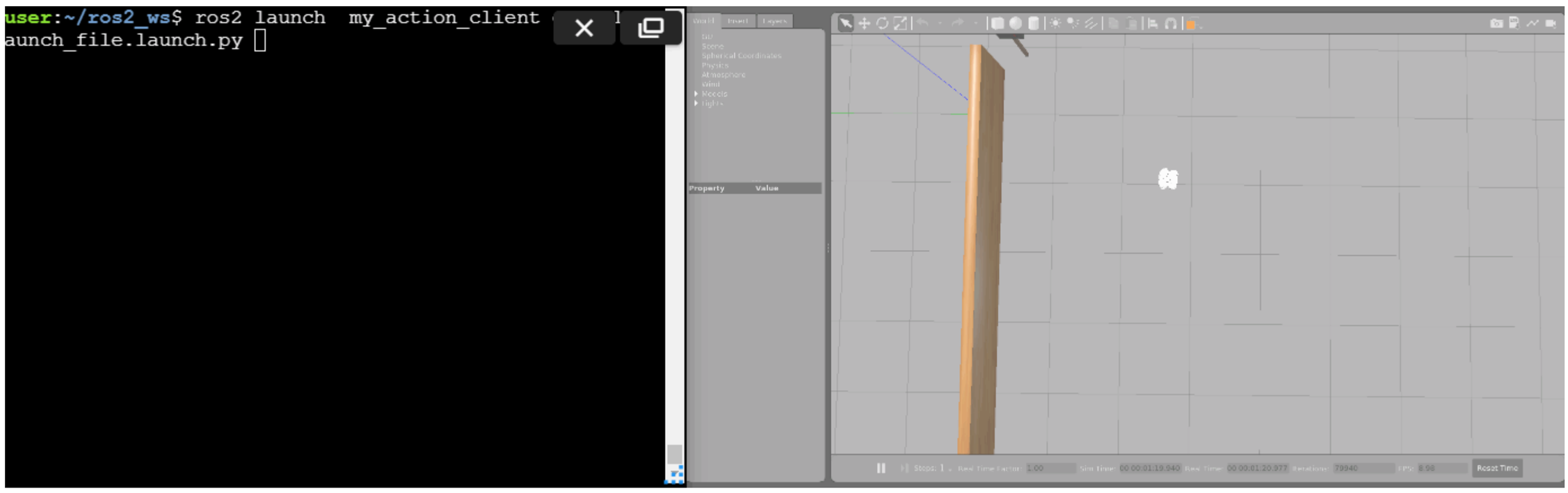
In [ ]:

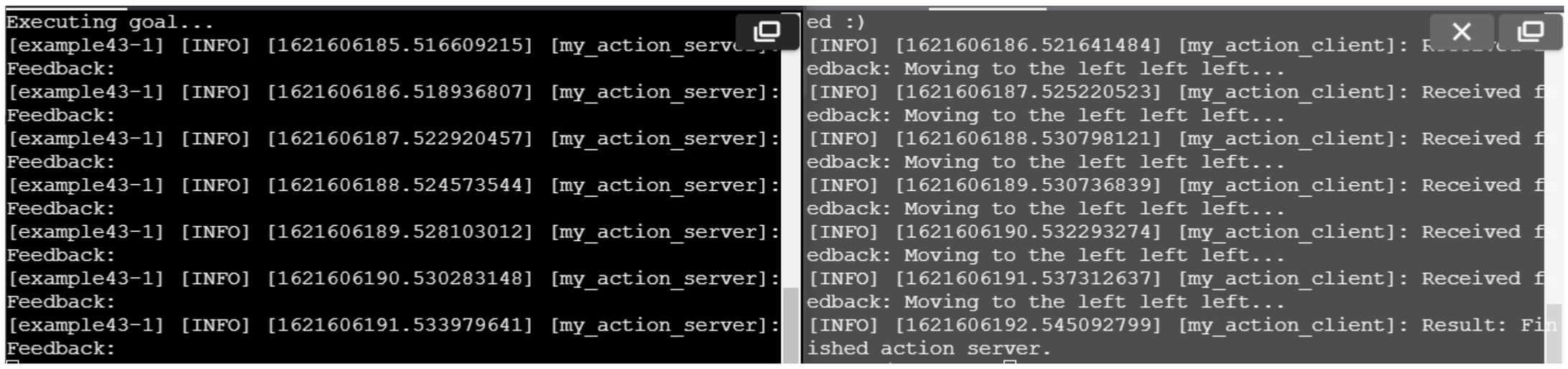
```
colcon build
```

In [ ]:

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

As you can see, everything is working fine again.





6.5 Actions Quiz

To evaluate this quiz, we will ask you to perform various tasks. For each task, **precise instructions** are provided: name of the package, 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 is different from the name specified in the exam instructions, your exercise will be marked as **FAILED**, even though it works correctly.

Create an Action Server with a custom Action Interface to compute the distance moved by the robot

- The new Action Server will receive an integer as a goal. This integer will define the number of **seconds** during which the Server will record the total distance traveled by the robot and publish it into a Topic named `/total_distance`.
- Suppose the Action Server receives a number 20 as a goal. In that case, the Server will monitor the total distance traveled by the robot during 20 seconds and publish this distance into the Topic `/total_distance`.
- When the time passes, it will return the final **total distance** traveled to the Client, and it will stop publishing into the Topic.
- As feedback, the Action Server publishes the **current distance** traveled by the robot every second.
- When the Action finishes, the result will return the following:
  - A boolean with `True`
  - The total distance traveled by the robot

TO DO

- Create a new package named **actions_quiz_msg**, where you will add the new action interface.

NOTE: This package will ONLY contain the action custom interface.

- Notes -

Useful Data for the Quiz:

- You need to create a new action message with the following structure:

In [ ]:

```
int32 seconds
---
bool status
float64 total_dist
---
float64 current_dist
```

- End of Notes -

- Create another package named **actions_quiz**. In this package, place the Python scripts containing the Actions Server and Client, and the launch files to start them.
- Create the Python script that will contain the Action Server code.
- Use the data received in the `seconds` variable to create a loop. Inside this loop, do the following:
  - Publish the **current distance** traveled by the robot to a Topic named `/total_distance`.
  - Publish the **current distance** traveled by the robot as feedback of the Action.
- When the loop finishes (time has passed) return the result message containing the following:
  - The final **total distance** traveled by the robot.
  - A boolean as `True`
- Create a new launch file, named **actions_quiz_server.launch.py**, that launches the new Action.
- Test that when calling the `/distance_as` Action, the Action correctly computes the distance traveled by the robot. This means the distance value increases accordingly as the robot keeps moving.
- Test also that the Action publishes the **current distance** traveled value to the topic `/total_distance`.
- Create another Python script that contains the Action Client code. This Client calls the Action `/distance_as`, and makes the Action compute the traveled distance during **20 seconds**.
- Create the launch file that starts the Action Client and name it **actions_quiz_client.launch.py**.

- Notes -

Use the keyboard teleop node to move the robot and test that your Action works as expected:

Execute in Shell

In [ ]:

```
source /opt/ros/humble/setup.bash
source /home/simulations/ros2_sims_ws/install/setup.bash
ros2 run turtlebot3_teleop teleop_keyboard
```

- End of Notes -

- Notes -

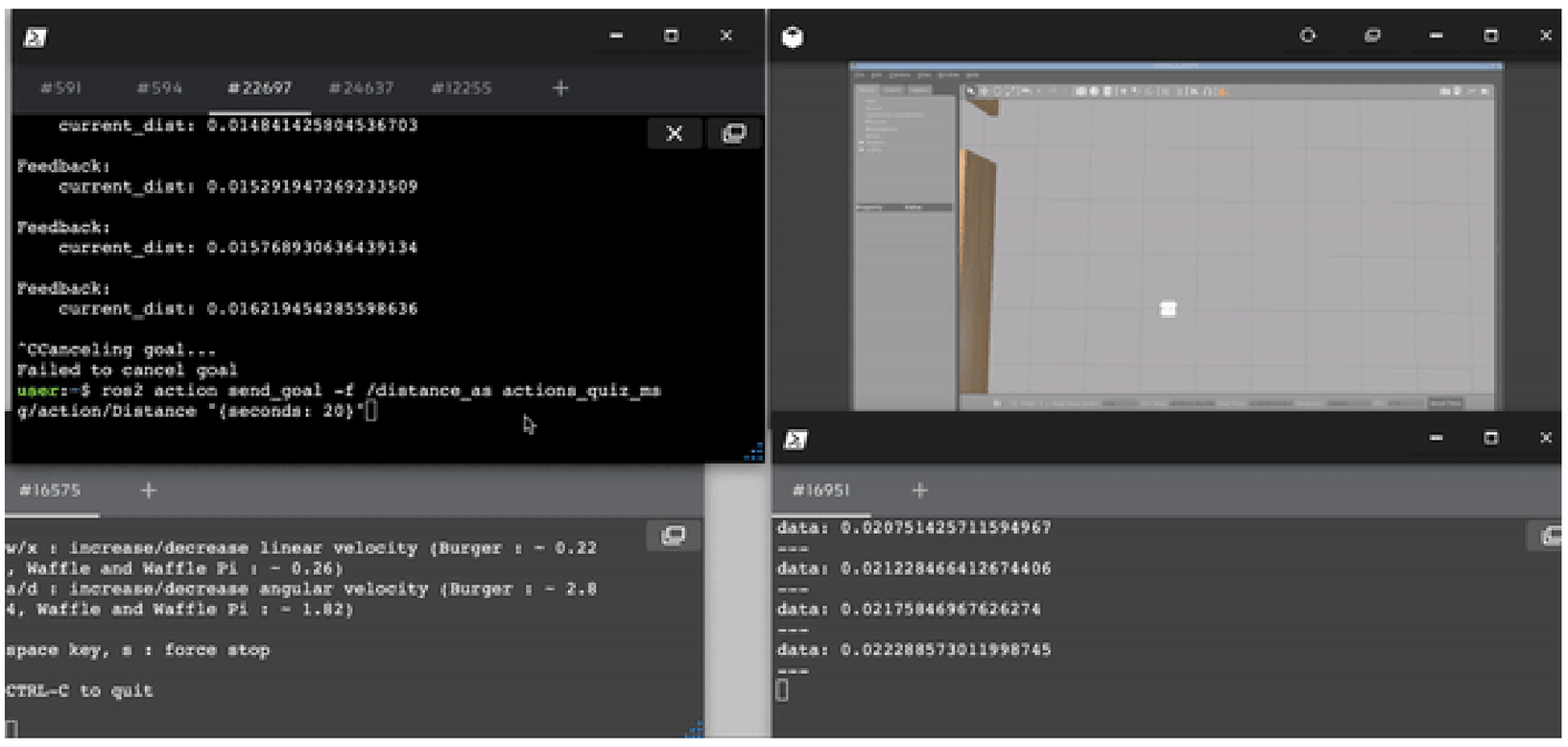
Check the following animated image as an example. You send a goal to the Action Server with the following command:

In [ ]:

```
ros2 action send_goal -f /distance_as actions_quiz_msg/action/Distance "{seconds: 20}"
```

Then, moving the robot using the `keyboard teleop`, you can observe the following:

- The current distance traveled at the `/total_distance` Topic.
- The current distance traveled as feedback in the Client.
- When the time finishes, the Topic publication stops and you get the final distance traveled as a result.



For reference, the shell output looks something like this:

```
Waiting for an Action Server to become available...
Sending goal:
     seconds: 20

Goal accepted with ID: f6cfc1b2342b4fcca3afab4ab358278f

Feedback:
    current_dist: 0.008692791680799913

Feedback:
    current_dist: 0.008692791680799913

Feedback:
    current_dist: 0.011565398269862739

Feedback:
    current_dist: 0.02643867417570701

Feedback:
    current_dist: 0.07284188343208137

Feedback:
    current_dist: 0.14743837934672827

Feedback:
    current_dist: 0.22504236973126862

Feedback:
    current_dist: 0.3121933785610431

Feedback:
    current_dist: 0.4020641216057754

Feedback:
    current_dist: 0.5172012277808455

Feedback:
    current_dist: 0.6203324689213692

Feedback:
    current_dist: 0.7318201633641616

Feedback:
    current_dist: 0.851611805791262

Feedback:
    current_dist: 0.9763074980595456

Feedback:
    current_dist: 1.1107200491493348

Feedback:
    current_dist: 1.2580806379613176

Feedback:
    current_dist: 1.4136289330121792

Feedback:
    current_dist: 1.5282445296809037

Feedback:
    current_dist: 1.6756092818491872

Feedback:
    current_dist: 1.8475377045475367

Result:
    status: true
total_dist: 2.0030961600503145

Goal finished with status: SUCCEEDED
```

- End of Notes -

Specifications

- The name of the package where you will place all the code related to the Action Server and Client will be **actions_quiz**.
- The name of the package where you will place all the code related to the custom Action Interface will be **actions_quiz_msg**.
- The name of the launch file that will start your Action Server will be **actions_quiz_server.launch.py**.
- The name of the launch file that will start your Action Client will be **actions_quiz_client.launch.py**.
- The name of the Action will be `/distance_as`.
- The name of your Action message file will be `Distance.action`.
- Before correcting your quiz, ensure you have terminated all the programs in your web shells.
- DO NOT use custom interfaces defined in external packages, for instance, in the `custom_interfaces` package.

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. Did the Action custom message compile successfully?
4. Was the Action Server started with the launch file?
5. Does the Action appear when listing the available Actions?
6. Did the Action Server correctly compute the robot's traveled distance?

The grader will provide feedback on any failed step so that 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

Do not get frustrated if you receive a low mark or fail the quiz! You will have the chance to retake the quiz to improve your score.

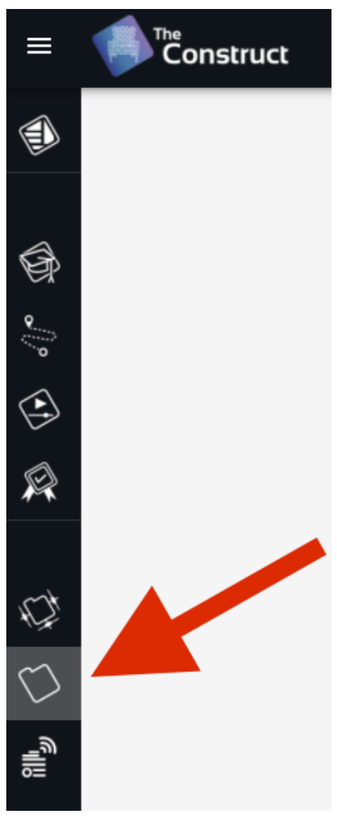
You are now ready to start the project for this course!

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.

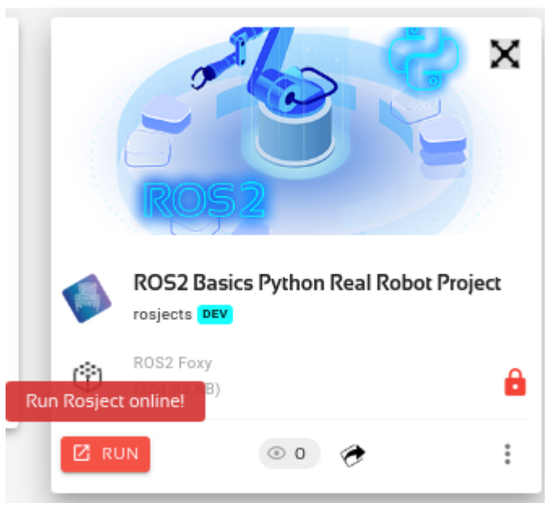
The ROSDS is included with your subscription and is integrated inside The Construct. So no extra effort needs to be made by you. Well, you will need to expend some extra learning effort! But that is why you are here!

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 III 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 III of the rosject before attempting the next unit in this course!

English proofread