



UNIVERSIDAD  
DE MÁLAGA

| **uma.es**

## S2 – PLAYING WITH ROS2

---

José Raúl Ruiz Sarmiento

Departamento de Ingeniería de Sistemas y Automática

# LECTURE CONTENTS

## 1. PLAYING WITH THE TERMINAL

Checking the installation, turtlesim, rqt, rqt\_graph.

## 2. DEVELOPING SOFTWARE

Colcon, Workspaces, Creating packages and nodes (CMake and python).

# 1. PLAYING WITH ROS 2

# 1. PLAYING WITH THE TERMINAL

## CHECKING THE INSTALLATION

- Let's start by checking that ROS2 exists...

*\$ ros2*

If this doesn't work, your in troubles!

ROS 2 includes a suite of command-line tools for introspecting a ROS 2 system, where the main entry point for the tools is the command *ros2*, which itself has various sub-commands for introspecting and working with nodes, topics, services, and more.

```
jotaraul@DESKTOP-3NMQR76: ~
jotaraul@DESKTOP-3NMQR76: ~ 80x35
jotaraul@DESKTOP-3NMQR76:~$ ros2
usage: ros2 [-h] [--use-python-default-buffering]
          Call `ros2 <command> -h` for more detailed usage. ...

ros2 is an extensible command-line tool for ROS 2.

options:
  -h, --help                show this help message and exit
  --use-python-default-buffering
                          Do not force line buffering in stdout and instead use
                          the python default buffering, which might be affected
                          by PYTHONUNBUFFERED/-u and depends on whatever stdout
                          is interactive or not

Commands:
  action      Various action related sub-commands
  bag         Various rosbag related sub-commands
  component   Various component related sub-commands
  daemon      Various daemon related sub-commands
  doctor      Check ROS setup and other potential issues
  interface   Show information about ROS interfaces
  launch      Run a launch file
  lifecycle   Various lifecycle related sub-commands
  multicast   Various multicast related sub-commands
  node        Various node related sub-commands
  param       Various param related sub-commands
  pkg         Various package related sub-commands
  run         Run a package specific executable
  security    Various security related sub-commands
  service     Various service related sub-commands
  topic       Various topic related sub-commands
  wtf         Use `wtf` as alias to `doctor`

          Call `ros2 <command> -h` for more detailed usage.
jotaraul@DESKTOP-3NMQR76:~$
```

# 1. PLAYING WITH THE TERMINAL

## CHECKING THE INSTALLATION

- Now check that ROS2 environment variables are loaded:

```
$ printenv | grep ROS
```

This should display ROS 2-related environment variables, such as:

```
jotaraul@DESKTOP-3NMQR76:~$ printenv | grep ROS
ROS_VERSION=2
ROS_PYTHON_VERSION=3
ROS_LOCALHOST_ONLY=0
ROS_DISTRO=humble
```

**Note:** If nothing is displayed, then something happened during the installation. These variables are loaded automatically each time a new terminal is created, since you added this line to your `.bashrc` file:

```
$ echo "source /opt/ros/humble/setup.bash" >> ~/.bashrc
```

To load these variables manually:

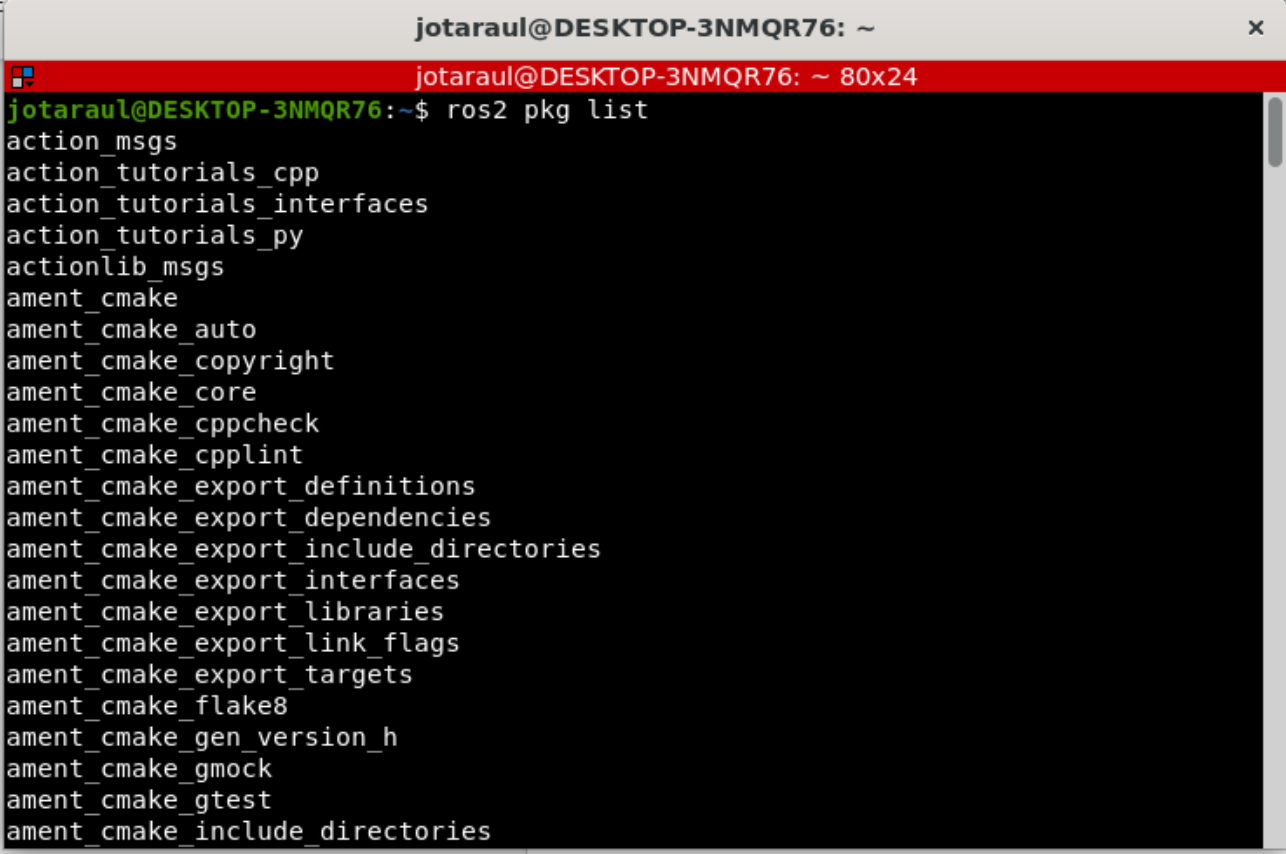
```
$ source /opt/ros/humble/setup.bash
```

# 1. PLAYING WITH THE TERMINAL

## CHECKING THE INSTALLATION

- Run the following command to check basic ROS 2 CLI (Command Line Interface) functionality:

```
$ ros2 pkg list
```



```
jotaraul@DESKTOP-3NMQR76: ~  
jotaraul@DESKTOP-3NMQR76: ~ 80x24  
jotaraul@DESKTOP-3NMQR76:~$ ros2 pkg list  
action_msgs  
action_tutorials_cpp  
action_tutorials_interfaces  
action_tutorials_py  
actionlib_msgs  
ament_cmake  
ament_cmake_auto  
ament_cmake_copyright  
ament_cmake_core  
ament_cmake_cppcheck  
ament_cmake_cpplint  
ament_cmake_export_definitions  
ament_cmake_export_dependencies  
ament_cmake_export_include_directories  
ament_cmake_export_interfaces  
ament_cmake_export_libraries  
ament_cmake_export_link_flags  
ament_cmake_export_targets  
ament_cmake_flake8  
ament_cmake_gen_version_h  
ament_cmake_gmock  
ament_cmake_gtest  
ament_cmake_include_directories
```

# 1. PLAYING WITH THE TERMINAL

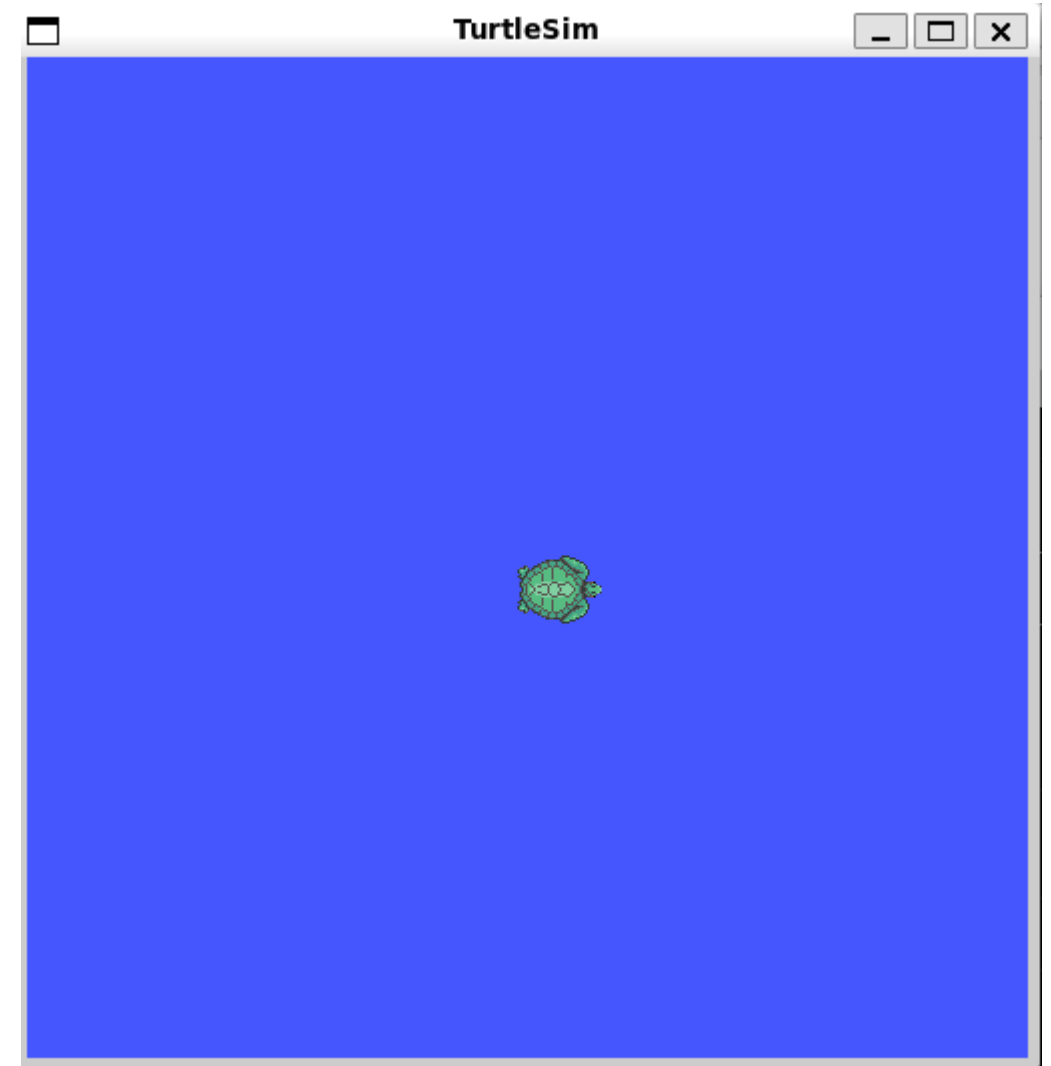
## TURTLESIM

- Now run `Turtlesim`, a lightweight simulator for learning ROS 2. It illustrates what ROS 2 does at the most basic level, to give you an idea of what you will do with a real robot or robot simulation later on.

```
$ ros2 run turtlesim  
  turtlesim_node
```

```
jotaraul@DESKTOP-3NMQR76:~$ ros2 run turtlesim turtlesim_node  
[INFO] [1733068966.289708926] [turtlesim]: Starting turtlesim with  
node name /turtlesim  
[INFO] [1733068966.297258044] [turtlesim]: Spawning turtle [turtle1]  
] at x=[5.544445], y=[5.544445], theta=[0.000000]
```

Default turtle's name



# 1. PLAYING WITH THE TERMINAL

## TURTLESIM

- We are going to teleoperate the turtle, so launch the following node:

```
$ ros2 run turtlesim turtle_teleop_key
```

- Ok we now have two nodes running. Which will be the available topics? Run in a new terminal:

```
$ ros2 topic list
```

- Move the robot and check the motion commands that are being sent with:

```
$ ros2 topic echo /turtle1/cmd_vel
```

This terminal should be active to control the turtle!

```
jotaraul@DESKTOP-3NMQR76:~$ ros2 run turtlesim turtle_teleop_key
Reading from keyboard
-----
Use arrow keys to move the turtle.
Use G|B|V|C|D|E|R|T keys to rotate to absolute orientations.
'F' to cancel a rotation.
'Q' to quit.
```

```
jotaraul@DESKTOP-3NMQR76:~$ ros2 topic list
/parameter_events
/rosout
/turtle1/cmd_vel
/turtle1/color_sensor
/turtle1/pose
```

```
jotaraul@DESKTOP-3NMQR76:~$ ros2 topic echo /turtle1/cmd_vel
linear:
  x: 2.0
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 0.0
---
```



# 1. PLAYING WITH THE TERMINAL

## TURTLESIM

- As you know, in ROS2 the basic communication mechanism are topics. Let's publish a motion command directly using CLI tools:

```
$ ros2 topic pub /turtle1/cmd_vel geometry_msgs/msg/Twist "{linear: {x: 2.0},  
angular: {z: 1.0}}"
```

### Explanation of the Command:

- `/turtle1/cmd_vel`: The topic controlling the turtle's motion.
- `geometry_msgs/msg/Twist`: The message type used to define linear and angular velocity.
- `{linear: {x: 2.0}, angular: {z: 1.0}}`: The specific velocity values being sent:
  - `linear.x = 2.0`: Moves the turtle forward at a speed of 2 units/sec.
  - `angular.z = 1.0`: Rotates the turtle counterclockwise at a speed of 1 radian/sec.
- You can adjust `linear.x` and `angular.z` to control the turtle's speed and direction.

- Try different velocities and, finally, **stop the robot!**

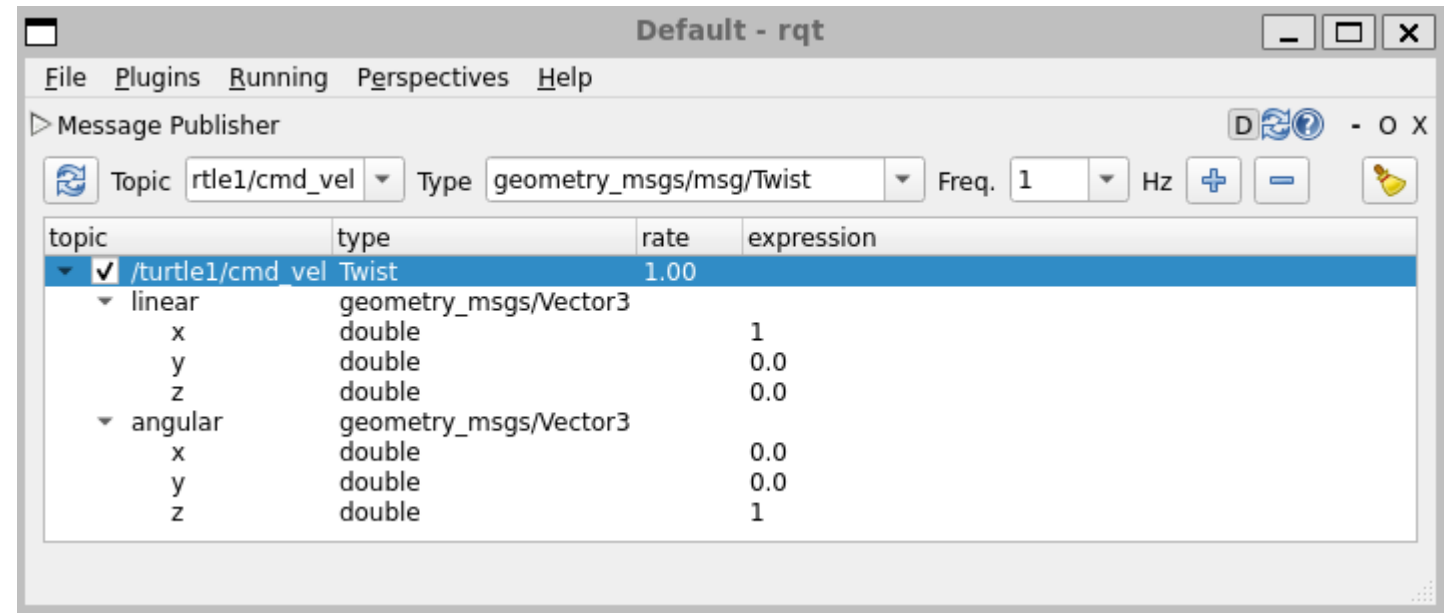
# 1. PLAYING WITH THE TERMINAL

## RQT

It may take some time for rqt to locate all the plugins. If you click on Plugins but don't see Services or any other options, you should close rqt and enter the command `rqt --force-discover` in your terminal.

- To be publishing commands, calling services and actions, etc., through CLI... is a pain on the neck!
- Fortunately ROS2 provides `rqt`, a graphical user interface (GUI) tool. Everything done in rqt can be done on the command line, but rqt provides a more user-friendly way to manipulate ROS 2 elements. Do the following:

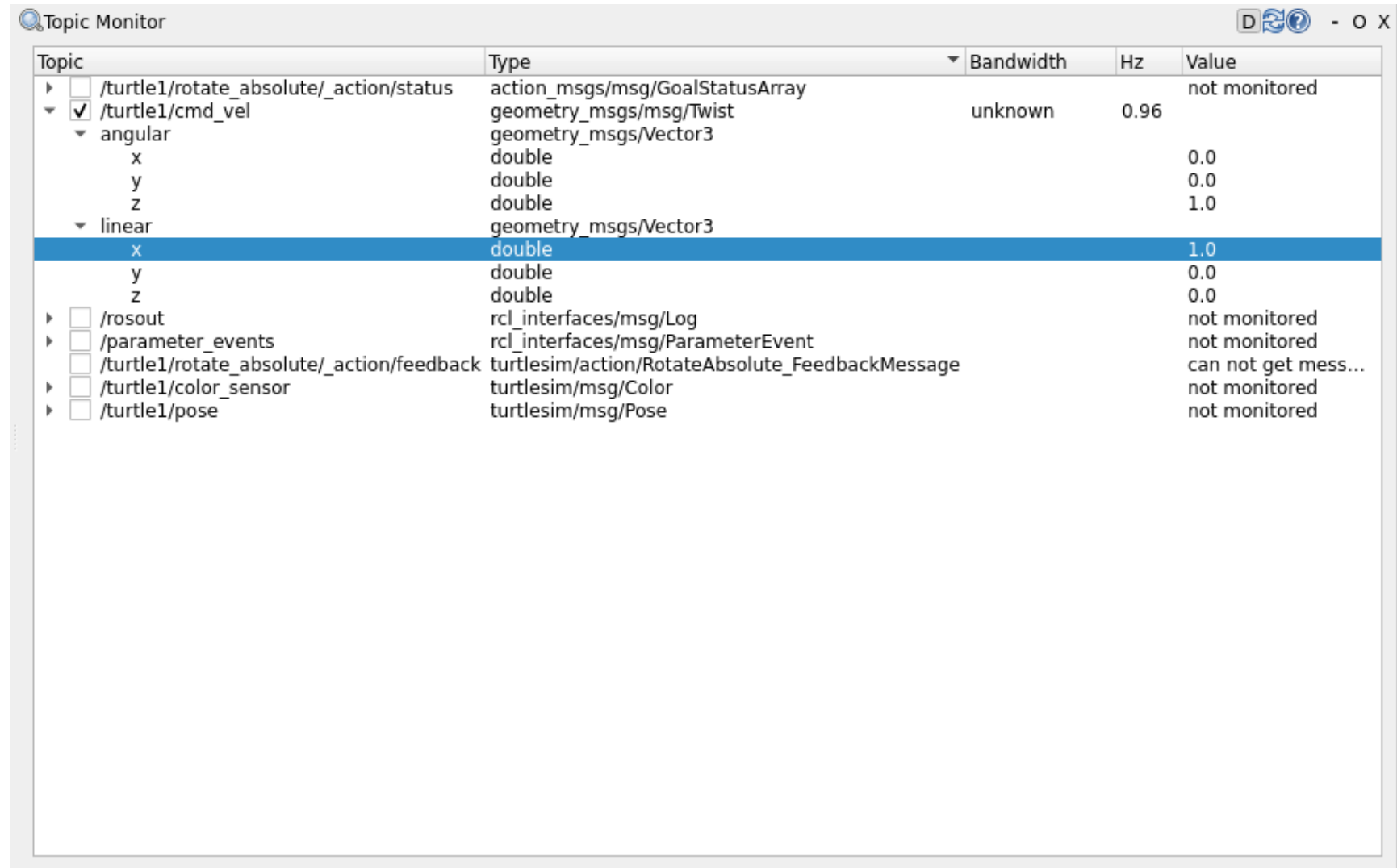
1. Run `$ rqt`.
2. Click on Plugins > Topics > Message publisher.
3. Select `/turtle1/cmd_vel`.
4. Click the "+" icon (Add new publisher) and edit the message.
5. Right click Public selected once.
6. Mark the box to keep publishing the message.



# 1. PLAYING WITH THE TERMINAL

## RQT

- There is another plugin called Topic monitor! Open it and explore the values of published messages on topics and statistics like bandwidth, frequency of publication (Hz), etc.

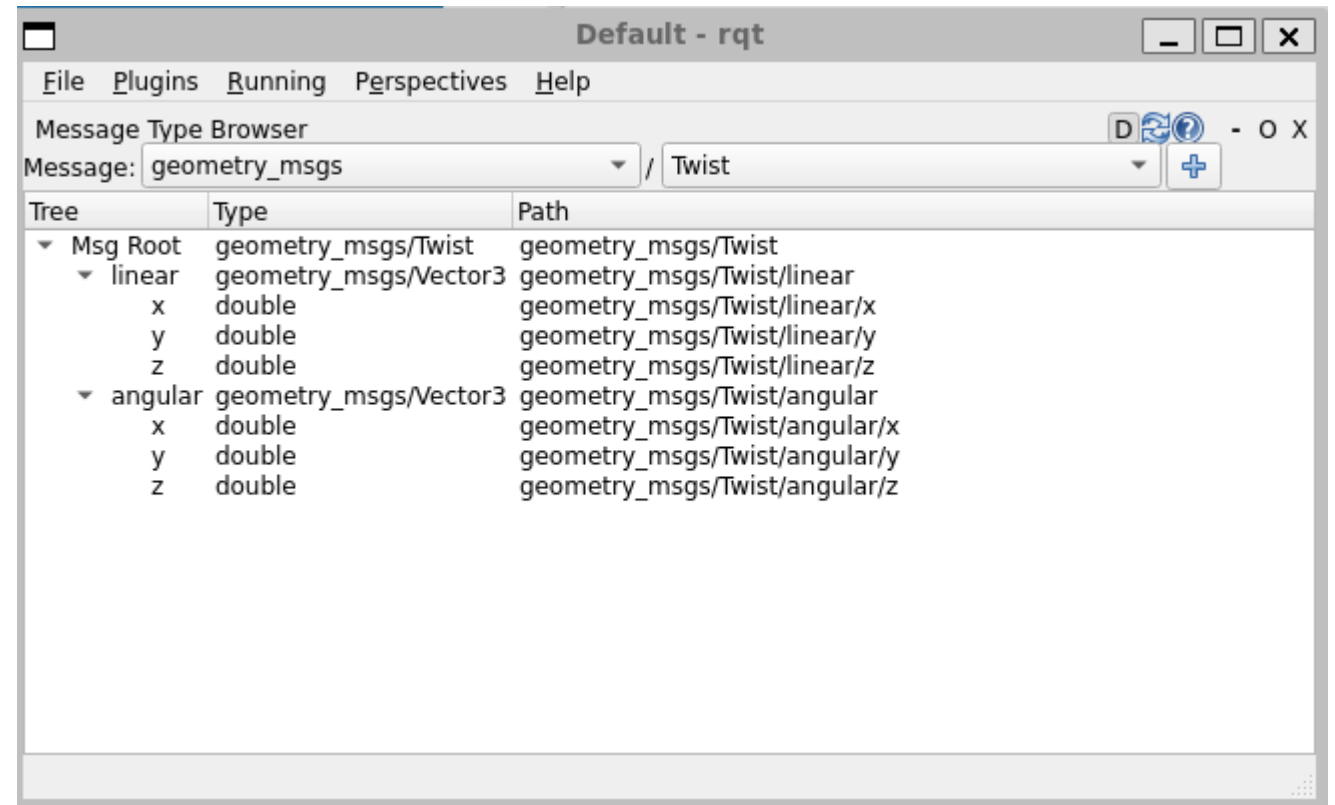


Topic	Type	Bandwidth	Hz	Value
<input type="checkbox"/> /turtle1/rotate_absolute/_action/status	action_msgs/msg/GoalStatusArray			not monitored
<input checked="" type="checkbox"/> /turtle1/cmd_vel	geometry_msgs/msg/Twist	unknown	0.96	
<input type="checkbox"/> angular	geometry_msgs/Vector3			
x	double			0.0
y	double			0.0
z	double			1.0
<input type="checkbox"/> linear	geometry_msgs/Vector3			
x	double			1.0
y	double			0.0
z	double			0.0
<input type="checkbox"/> /rosout	rcl_interfaces/msg/Log			not monitored
<input type="checkbox"/> /parameter_events	rcl_interfaces/msg/ParameterEvent			not monitored
<input type="checkbox"/> /turtle1/rotate_absolute/_action/feedback	turtlesim/action/RotateAbsolute_FeedbackMessage			can not get mess...
<input type="checkbox"/> /turtle1/color_sensor	turtlesim/msg/Color			not monitored
<input type="checkbox"/> /turtle1/pose	turtlesim/msg/Pose			not monitored

# 1. PLAYING WITH THE TERMINAL

## RQT

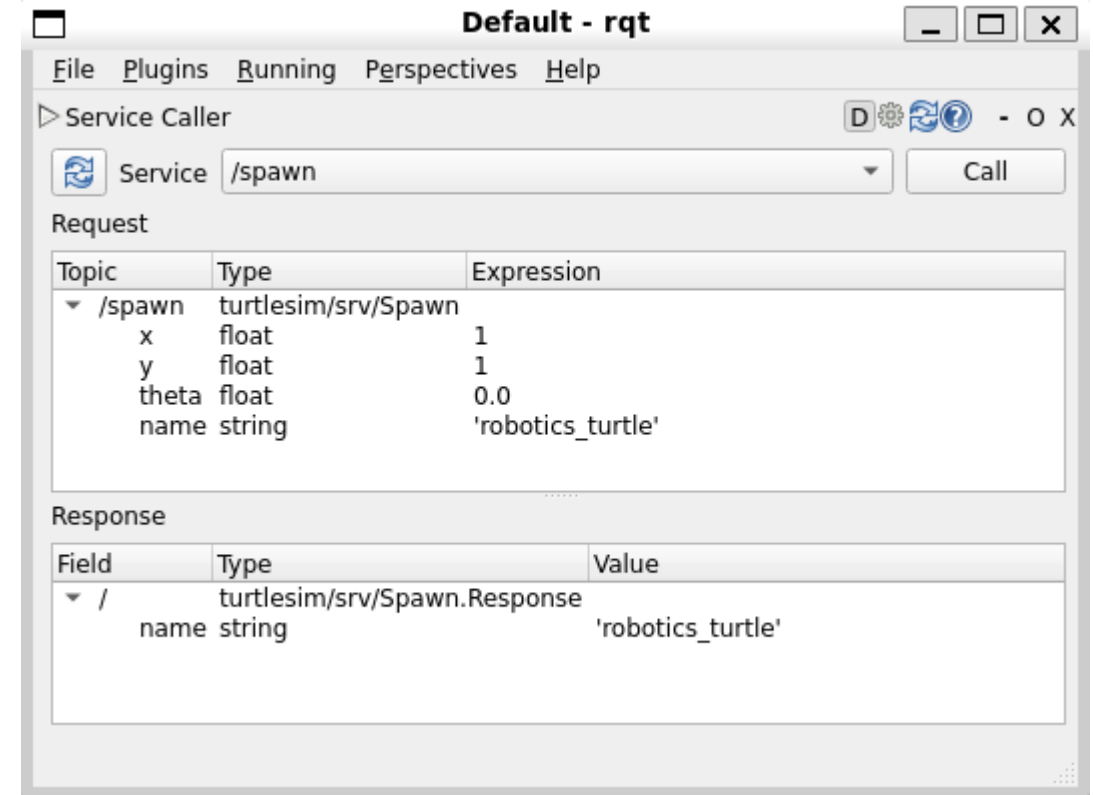
- And finally, play a bit with the Message Type Browser one, which permits you to inspect the messages' definition. So useful!



# 1. PLAYING WITH THE TERMINAL

## RQT

- Now, let's call a service from this nice GUI. For that:
  1. Open Plugins > Services > Service Caller.
  2. Select the /spawn service.
  3. Edit the message with a unique name and position.
  4. Click on "Call".
- If you refresh the services' list, you will see now some appearing with the name of your turtle.



# 1. PLAYING WITH THE TERMINAL

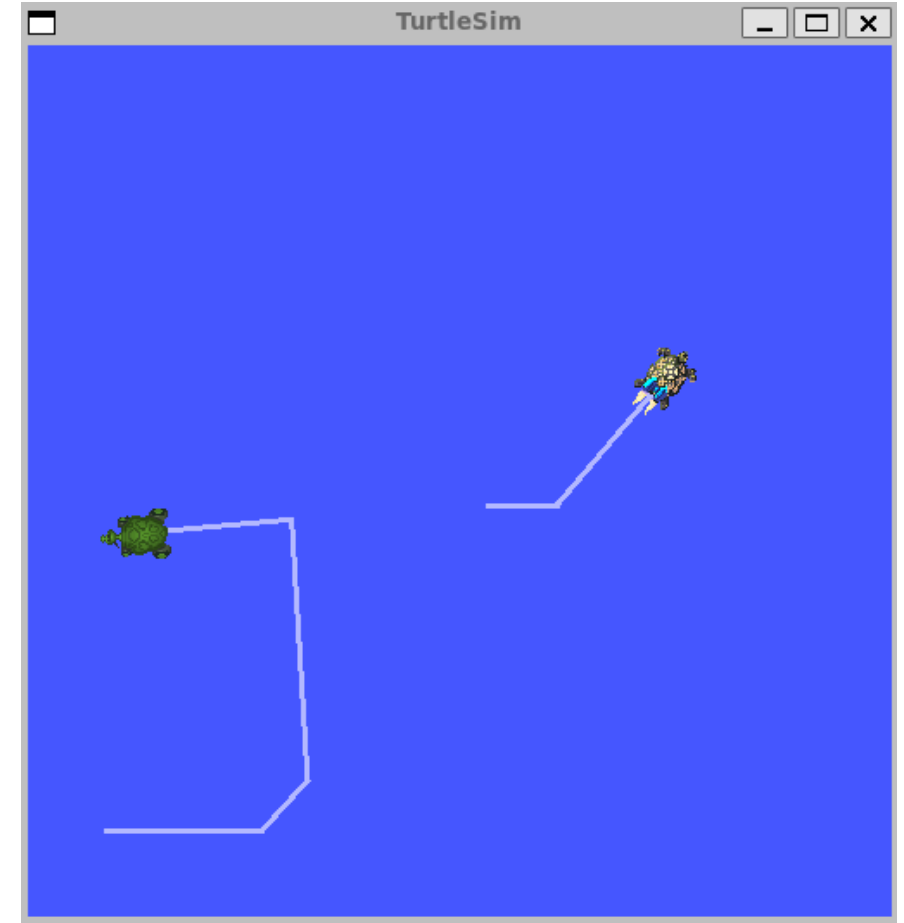
## TURTLESIM

- You need a second teleop node in order to control turtle2 (or the name you used). However, if you try to run the same command as before, you will notice that this one also controls turtle1. The way to change this behavior is by remapping the `cmd_vel` topic.

In a new terminal, run:

```
ros2 run turtlesim turtle_teleop_key --ros-args --remap  
turtle1/cmd_vel:=turtle2/cmd_vel
```

Now, you can move turtle2 when this terminal is active, and turtle1 when the other terminal running `turtle_teleop_key` is active.



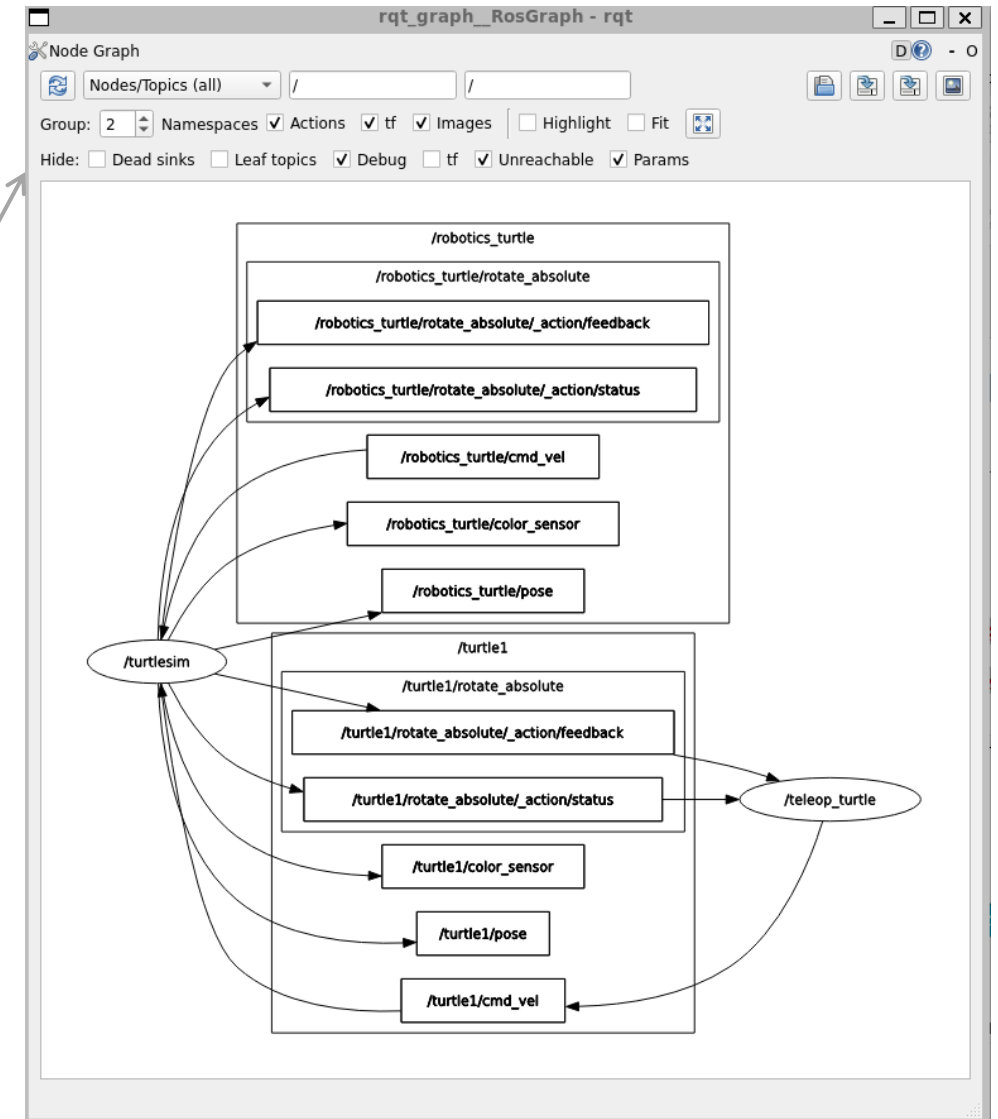
# 1. PLAYING WITH THE TERMINAL

## RQT GRAPH

- Finally, let's launch `rqt_graph` to visually check how these nodes interact.

```
$ rqt_graph
```

Within the `rqt_graph` interface you can play with the items to show/hide



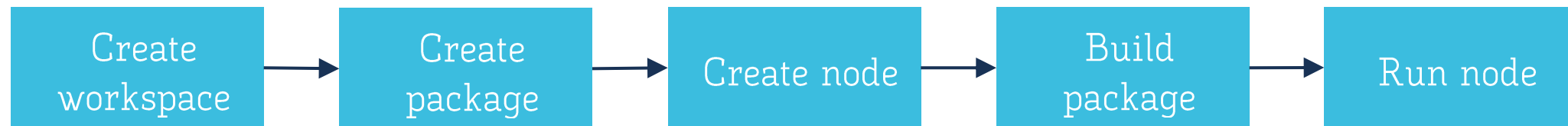
## 2. ROS 2 – DEVELOPING SOFTWARE



## 2. ROS 2 – DEVELOPING SOFTWARE

### COLCON

- When you become a [ROS developer](#), you aim to implement [new nodes](#), which are organized into [packages](#).
- [colcon](#): official build tool of ROS based on Cmake and python. It is in charge of:
  - managing dependences among nodes,
  - organizing and making packages accessible, and
  - generating executables from source code.
- Workflow (from workspace to node):



## 2. ROS 2 – DEVELOPING SOFTWARE

### COLCON WORKSPACE

- A **workspace** is a directory where colcon operates to build packages, with the following directories:
  - src: place to locate the source code of ROS 2 packages.
  - build: intermediate files are stored here.
  - install: contains the installation of each package.
  - log: contains diverse login information.
- The following commands create a workspace called `dev_ws` and its `/src` directory, move to it, calls to colcon build (this creates and initial set of directories and files), and list the resulting directories. With this we get a workspace ready to work with!

```
File Edit View Search Terminal Tabs Help
robotics@humbl... x robotics@humbl... x robotics@humbl... x robotics@humbl... x
robotics@humble:~$ mkdir -p ~/dev_ws/src
robotics@humble:~$ cd ~/dev_ws/
robotics@humble:~/dev_ws$ colcon build

Summary: 0 packages finished [0.30s]
robotics@humble:~/dev_ws$ ls
build install log src
robotics@humble:~/dev_ws$ echo "source ~/dev_ws/install/local_setup.bash" >> ~/.bashrc
robotics@humble:~/dev_ws$ source ~/.bashrc
```

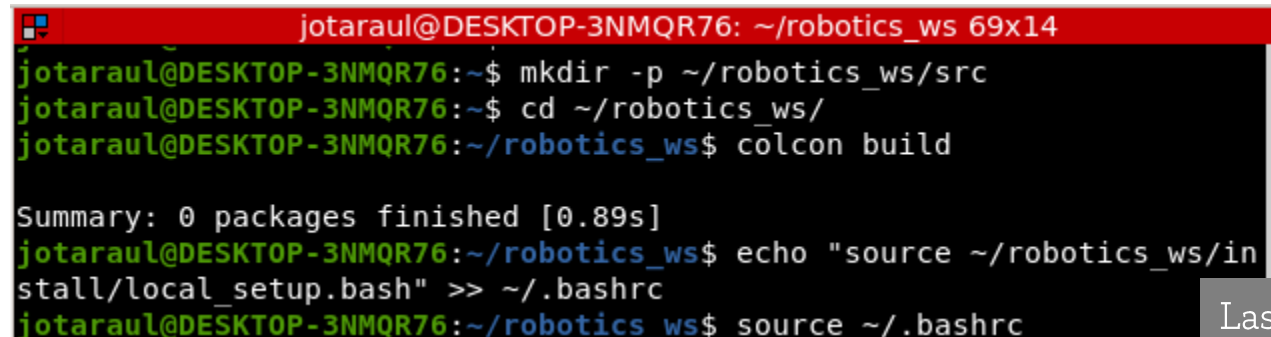
Last two lines add useful ROS 2 environment variables (e.g. where to look for installed executables) to bashrc. **MUST BE DONE ONLY ONCE!**

## 2. ROS 2 – DEVELOPING SOFTWARE

### COLCON WORKSPACE

- Let's create our first workspace! For that use the following commands:

```
$ mkdir -p ~/robotics_ws/src
$ cd ~/robotics_ws/
$ colcon build
$ ls
$ echo "source ~/robotics_ws/install/local_setup.bash" >> ~/.bashrc
$ source ~/.bashrc
```

A terminal window with a red title bar showing the user 'jotaraul' on a machine named 'DESKTOP-3NMQR76'. The terminal displays the execution of the commands: 'mkdir -p ~/robotics\_ws/src', 'cd ~/robotics\_ws/', 'colcon build', 'echo "source ~/robotics\_ws/install/local\_setup.bash" >> ~/.bashrc', and 'source ~/.bashrc'. The output of 'colcon build' shows 'Summary: 0 packages finished [0.89s]'.

```
jotaraul@DESKTOP-3NMQR76: ~/robotics_ws 69x14
jotaraul@DESKTOP-3NMQR76:~$ mkdir -p ~/robotics_ws/src
jotaraul@DESKTOP-3NMQR76:~$ cd ~/robotics_ws/
jotaraul@DESKTOP-3NMQR76:~/robotics_ws$ colcon build

Summary: 0 packages finished [0.89s]
jotaraul@DESKTOP-3NMQR76:~/robotics_ws$ echo "source ~/robotics_ws/in
stall/local_setup.bash" >> ~/.bashrc
jotaraul@DESKTOP-3NMQR76:~/robotics_ws$ source ~/.bashrc
```

Last two lines add useful ROS 2 environment variables (e.g. where to look for installed executables) to bashrc. **MUST BE DONE ONLY ONCE!**

## 2. ROS 2 – DEVELOPING SOFTWARE

### CREATING YOUR OWN PACKAGE

- Typical structure of the workspace:

```
workspace_folder/  
  src/  
    package_1/  
      CMakeLists.txt      -- CMakeLists.txt file for package_1  
      package.xml         -- Package manifest for package_1  
  
    package_2/  
      setup.py            -- Installation instructions for package_2  
      package.xml         -- Package manifest for package_2  
      resource/package_2  
  
    ...  
  
    package_n/  
      CMakeLists.txt      -- CMakeLists.txt file for package_1  
      package.xml         -- Package manifest for package_n
```

#### Remind:

- A single workspace can contain as many packages as you want.
- A package can have different build types (e.g. CMake or Python).
- No nested packages are allowed.

#### Remind:

- To build a package, just go to the workspace root directory and call colcon with `colcon build`.
- To build only package\_1: `colcon build --packages-select package_1`

## 2. ROS 2 – DEVELOPING SOFTWARE

### CREATING YOUR OWN PACKAGE

- `ros2 pkg create` command, with **CMake**:

More convenient when  
developing C++ nodes

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

This creates a directory with the same name and  
some initial content:

```
File Edit View Search Terminal Help
robotics@humble:~/ros2_ws/src/my_package_cmake$ ls
CMakeLists.txt  include  package.xml  src
robotics@humble:~/ros2_ws/src/my_package_cmake$
```

In this case, as we called `ros2 pkg create` with  
the `-node-name` option, a node called `my_node`  
with a simple Hello World executable is created.

Run it only inside the `src` folder!!

```
File Edit View Search Terminal Help
robotics@humble:~/ros2_ws/src$ ros2 pkg create --build-type
ament_cmake --node-name my_node my_package_cmake
going to create a new package
package name: my_package_cmake
destination directory: /home/robotics/ros2_ws/src
package format: 3
version: 0.0.0
description: TODO: Package description
maintainer: ['robotics <humble@example.com>']
licenses: ['TODO: License declaration']
build type: ament_cmake
dependencies: []
node_name: my_node
creating folder ./my_package_cmake
creating ./my_package_cmake/package.xml
creating source and include folder
creating folder ./my_package_cmake/src
creating folder ./my_package_cmake/include/my_package_cmake
creating ./my_package_cmake/CMakeLists.txt
creating ./my_package_cmake/src/my_node.cpp

[WARNING]: Unknown license 'TODO: License declaration'. This
s has been set in the package.xml, but no LICENSE file has b
een created.
It is recommended to use one of the ament license identitife
rs:
Apache-2.0
BSL-1.0
```

## 2. ROS 2 – DEVELOPING SOFTWARE

### IMPLEMENTING YOUR OWN NODE (CMAKE)

1. Create a new source file in the src folder of the package. For example, `my_node.cpp`
2. Modify the `CMakeList.txt` at `/src/my_package_cmake/` to include a reference to a such file. This tells the ROS 2 system we want to have a new executable (node).

```
8 # find dependencies
9 find_package(ament_cmake REQUIRED)
10 find_package(rclcpp REQUIRED)
11 find_package(std_msgs REQUIRED)
12
13 add_executable(my_node src/my_node.cpp)
14 ament_target_dependencies(my_node rclcpp std_msgs)
15
16 install(TARGETS my_node
17   DESTINATION lib/${PROJECT_NAME})
```

3. Update `package.xml`. Add the necessary build, execution and testing dependencies
4. Develop the node itself. In our example, implement in `my_node.cpp` the `desiserd` functionality. You can use a simple text editor or an IDE (e.g. Visual Studio Code).
5. Compile the source code and generate the node executable.

```
$ ~/ros2_ws
$ colcon build
```

5. Run your node! `ros2 run my_package_cmake my_node` →

Depending on your configuration, you may previously need to go to your workspace root directory and source the setup files:

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

## 2. ROS 2 – DEVELOPING SOFTWARE

### CREATING YOUR OWN PACKAGE

- `ros2 pkg create` command, with **Python**:

More convenient when  
developing python nodes

```
ros2 pkg create --build-type ament_python  
<package_name>
```

This creates a directory with the same name and  
some initial content:

```
File Edit View Search Terminal Help
robotics@humble:~/ros2_ws/src/my_package_python$ ls
my_package_python  resource  setup.py
package.xml        setup.cfg test
robotics@humble:~/ros2_ws/src/my_package_python$
```

In this case, as we called `ros2 pkg create` with  
the `-node-name` option, a node called `my_node`  
with a simple Hello World executable is created.

Run it only inside the `src` folder!!

```
File Edit View Search Terminal Help
Either remove the directory or choose a different destination directory or package name
robotics@humble:~/ros2_ws/src$ ros2 pkg create --build-type ament_python --node-name my_node my_package_python
going to create a new package
package name: my_package_python
destination directory: /home/robotics/ros2_ws/src
package format: 3
version: 0.0.0
description: TODO: Package description
maintainer: ['robotics <humble@example.com>']
licenses: ['TODO: License declaration']
build type: ament_python
dependencies: []
node_name: my_node
creating folder ./my_package_python
creating ./my_package_python/package.xml
creating source folder
creating folder ./my_package_python/my_package_python
creating ./my_package_python/setup.py
creating ./my_package_python/setup.cfg
creating folder ./my_package_python/resource
creating ./my_package_python/resource/my_package_python
creating ./my_package_python/my_package_python/__init__.py
creating folder ./my_package_python/test
creating ./my_package_python/test/test_copyright.py
creating ./my_package_python/test/test_flake8.py
creating ./my_package_python/test/test_pep257.py
creating ./my_package_python/my_package_python/my_node.py
```



## 2. ROS 2 – DEVELOPING SOFTWARE

### CREATING YOUR OWN PACKAGE

- Ok, let's create our first package! We will call it `turtlesim_control`.
- For that, in the `src` folder, execute the following command:

```
$ ros2 pkg create --build-type  
ament_python turtlesim_control
```

- You must now have a directory with the package name and this initial content:

```
jotaraul@DESKTOP-3NMQR76:~/robotics_ws/src$ ls turtlesim_control/  
package.xml  resource  setup.cfg  setup.py  _test  turtlesim_control
```

```
jotaraul@DESKTOP-3NMQR76:~/robotics_ws/src$ ros2 pkg create --build-type  
ament_python turtlesim_control  
going to create a new package  
package name: turtlesim_control  
destination directory: /home/jotaraul/robotics_ws/src  
package format: 3  
version: 0.0.0  
description: TODO: Package description  
maintainer: ['jotaraul <jotaraul@todo.todo>']  
licenses: ['TODO: License declaration']  
build type: ament_python  
dependencies: []  
creating folder ./turtlesim_control  
creating ./turtlesim_control/package.xml  
creating source folder  
creating folder ./turtlesim_control/turtlesim_control  
creating ./turtlesim_control/setup.py  
creating ./turtlesim_control/setup.cfg  
creating folder ./turtlesim_control/resource  
creating ./turtlesim_control/resource/turtlesim_control  
creating ./turtlesim_control/turtlesim_control/__init__.py  
creating folder ./turtlesim_control/test  
creating ./turtlesim_control/test/test_copyright.py  
creating ./turtlesim_control/test/test_flake8.py  
creating ./turtlesim_control/test/test_pep257.py  
  
[WARNING]: Unknown license 'TODO: License declaration'. This has been se  
t in the package.xml, but no LICENSE file has been created.  
It is recommended to use one of the ament license identitifers:  
Apache-2.0  
BSL-1.0  
BSD-2.0  
BSD-2-Clause  
BSD-3-Clause  
GPL-3.0-only  
LGPL-3.0-only  
MIT  
MIT-0
```



## 2. ROS 2 – DEVELOPING SOFTWARE

### IMPLEMENTING YOUR OWN NODE (PYTHON)

1. Create a new python file in the `turtlesim_control` folder of the package called `turtlesim_control_node.py`

```
jotaraul@DESKTOP-3NMQR76:~/robotics_ws/src/turtlesim_control/turtlesim_control$ ls
__init__.py
jotaraul@DESKTOP-3NMQR76:~/robotics_ws/src/turtlesim_control/turtlesim_control$ touch turtlesim_control_node.py
jotaraul@DESKTOP-3NMQR76:~/robotics_ws/src/turtlesim_control/turtlesim_control$ ls
__init__.py  turtlesim_control_node.py
```

2. Modify `setup.py` at `/src/turtlesim_control/` by updating the maintainer, email, description, and license (put the same as in `package.xml`) and add a new entry telling ROS 2 about the existence of your node:

```
16     maintainer='jotaraul',
17     maintainer_email='jotaraul@todo.todo',
18     description='TODO: Package description',
19     license='TODO: License declaration',
20     tests_require=['pytest'],
21     entry_points={
22         'console_scripts': [
23             'turtlesim_control_node = turtlesim_control.turtlesim_control_node:main',
24         ],
25     },
```

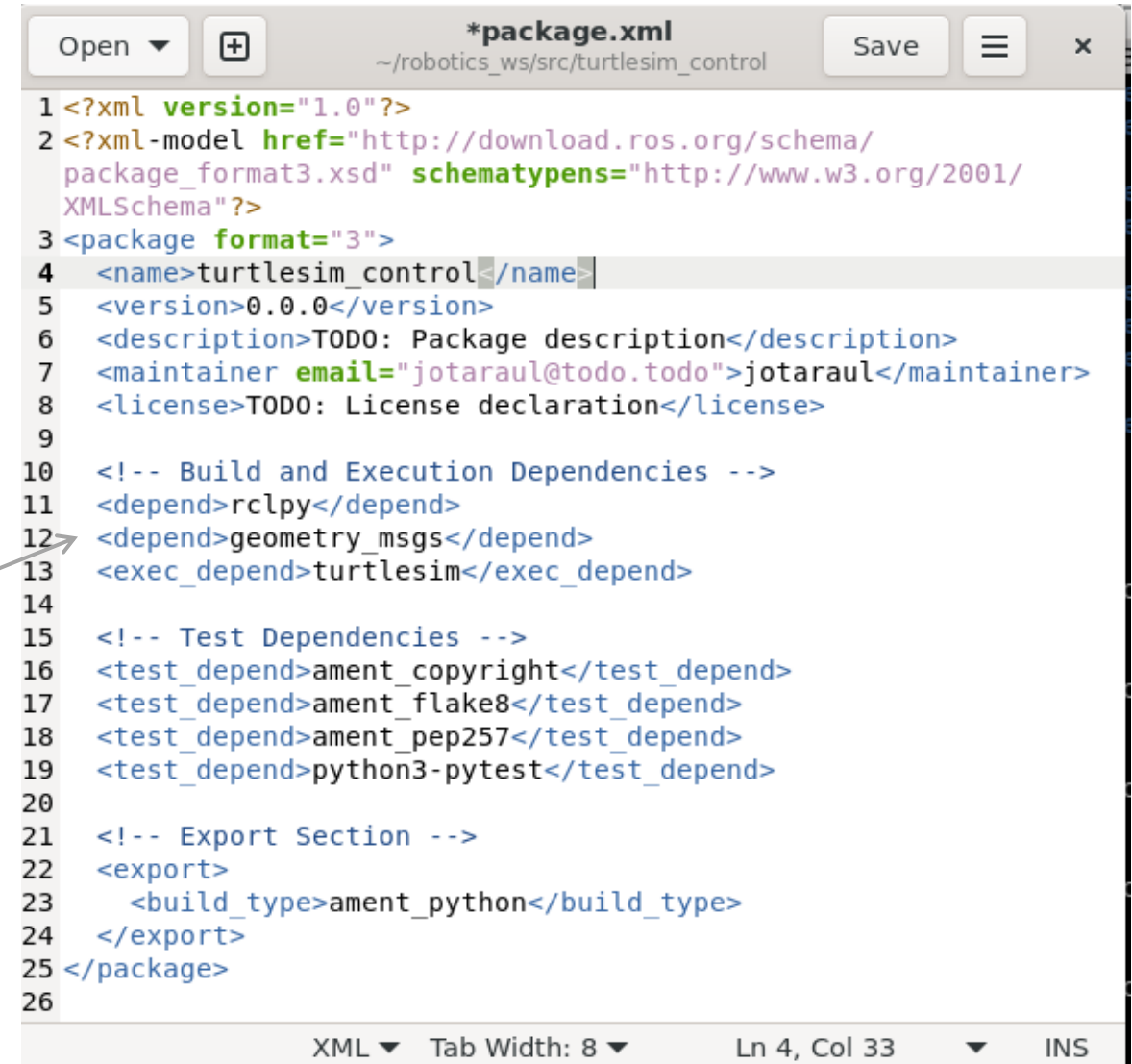
## 2. ROS 2 – DEVELOPING SOFTWARE

### IMPLEMENTING YOUR OWN NODE (PYTHON)

3. Update `package.xml`. Add the necessary dependencies for `geometry_msgs` and `turtlesim`.

- Dependency types:
  - `<build_depend>`,
  - `<exec_depend>`,
  - `<test_depend>`
  - `<depend>` (includes all the above)

We need to include these three



```
*package.xml
~/robotics_ws/src/turtlesim_control

1 <?xml version="1.0"?>
2 <?xml-model href="http://download.ros.org/schema/package_format3.xsd" schematypens="http://www.w3.org/2001/XMLSchema"?>
3 <package format="3">
4   <name>turtlesim_control</name>
5   <version>0.0.0</version>
6   <description>TODO: Package description</description>
7   <maintainer email="jotaraul@todo.todo">jotaraul</maintainer>
8   <license>TODO: License declaration</license>
9
10  <!-- Build and Execution Dependencies -->
11  <depend>rclpy</depend>
12  <depend>geometry_msgs</depend>
13  <exec_depend>turtlesim</exec_depend>
14
15  <!-- Test Dependencies -->
16  <test_depend>ament_copyright</test_depend>
17  <test_depend>ament_flake8</test_depend>
18  <test_depend>ament_pep257</test_depend>
19  <test_depend>python3-pytest</test_depend>
20
21  <!-- Export Section -->
22  <export>
23    <build_type>ament_python</build_type>
24  </export>
25 </package>
26
```

## 2. ROS 2 – DEVELOPING SOFTWARE

### IMPLEMENTING YOUR OWN NODE (PYTHON)

4. **Develop the node itself.** In our example, implement in `turtlesim_control_node.py` the desired functionality. You can use a simple text editor or an IDE (e.g. Visual Studio Code). Copy this code:

```
1 import rclpy
2 from rclpy.node import Node
3 from geometry_msgs.msg import Twist
4 from turtlesim.msg import Pose
5
6 class TurtlesimControlNode(Node):
7     def __init__(self):
8         super().__init__('turtlesim_control_node')
9
10        # Log message indicating the node has started
11        self.get_logger().info("Turtlesim Control Node is now running...")
12
13        # Subscriber to the /turtle1/pose topic
14        self.pose_subscription = self.create_subscription(
15            Pose, # Message type for the /turtle1/pose topic
16            '/turtle1/pose', # Topic name to subscribe to
17            self.pose_callback, # Callback function that processes the message
18            10 # QoS queue size
19        )
20        self.pose_subscription # Prevent unused variable warning
21
22        # Publisher to the /turtle1/cmd_vel topic
23        self.velocity_publisher = self.create_publisher(
24            Twist, # Message type for the /turtle1/cmd_vel topic
25            '/turtle1/cmd_vel', # Topic name to publish to
26            10 # QoS queue size
27        )
28
29        # Timer to publish velocity commands periodically
30        self.timer = self.create_timer(0.5, self.publish_velocity_command)
31
32        # Store the latest pose received from the subscriber
33        self.current_pose = None
```

Class constructor

Subscriber

Publisher

## 2. ROS 2 – DEVELOPING SOFTWARE

### IMPLEMENTING YOUR OWN NODE (PYTHON)

```
33 def pose_callback(self, msg):
34     """
35     Callback function for the /turtle1/pose topic.
36     Called whenever a new Pose message is received.
37     Parameters:
38     - msg (Pose): The message containing the turtle's position, orientation, and velocity.
39     """
40     self.current_pose = msg # Store the received pose in the node's state
41     # Print the pose information to the terminal
42     self.get_logger().info(
43         f'Turtle Pose - x: {msg.x:.2f}, y: {msg.y:.2f}, theta: {msg.theta:.2f}'
44     )
45
46 def publish_velocity_command(self):
47     """
48     Publishes velocity commands to the /turtle1/cmd_vel topic.
49     This method is periodically called by the timer.
50     """
51     if self.current_pose is not None: # Ensure we have a valid pose before publishing
52         # Create a Twist message to specify linear and angular velocities
53         twist = Twist()
54         twist.linear.x = 2.0 # Move forward at a speed of 2.0 units/sec
55         twist.angular.z = 1.0 # Rotate counterclockwise at a speed of 1.0 rad/sec
56
57         # Publish the velocity command to the /turtle1/cmd_vel topic
58         self.velocity_publisher.publish(twist)
59
60         # Log the command for debugging
61         self.get_logger().info(f'Published velocity command: {twist}')
62
```

## 2. ROS 2 – DEVELOPING SOFTWARE

### IMPLEMENTING YOUR OWN NODE (PYTHON)

```
64 def main(args=None):
65     """
66     Main function to start the node.
67     """
68     rclpy.init(args=args) # Initialize the ROS 2 system
69     turtlesim_control_node = TurtlesimControlNode() # Create an instance of the node
70     rclpy.spin(turtlesim_control_node) # Keep the node running until interrupted
71     turtlesim_control_node.destroy_node() # Clean up the node
72     rclpy.shutdown() # Shut down the ROS 2 system
73
74
75 if __name__ == '__main__':
76     main()
```

## 2. ROS 2 – DEVELOPING SOFTWARE

### IMPLEMENTING YOUR OWN NODE (PYTHON)

5. **Install python packages and set environment scripts.** This is done with the `colcon build` command, which carry out the following actions:

- **Installation Setup:** Python packages defined with `ament_python` (in `setup.py`) are installed into the `install/` directory. For example:
  - The `setup.py` entry points (e.g., `console_scripts`) are registered as executables in the workspace.
  - The package files are copied or symbolically linked into the `install/` directory.
- **Environment Scripts:** ROS 2 environment setup scripts are generated in the `install/` directory, which ensure the workspace dependencies and paths are properly configured.

Run:

```
$ ~/ros2_ws  
$ colcon build
```

```
jotaraul@DESKTOP-3NMQR76:~/robotics_ws/src$ cd ~/robotics_ws/  
jotaraul@DESKTOP-3NMQR76:~/robotics_ws$ colcon build  
Starting >>> turtlesim_control  
Finished <<< turtlesim_control [1.68s]  
  
Summary: 1 package finished [2.18s]
```

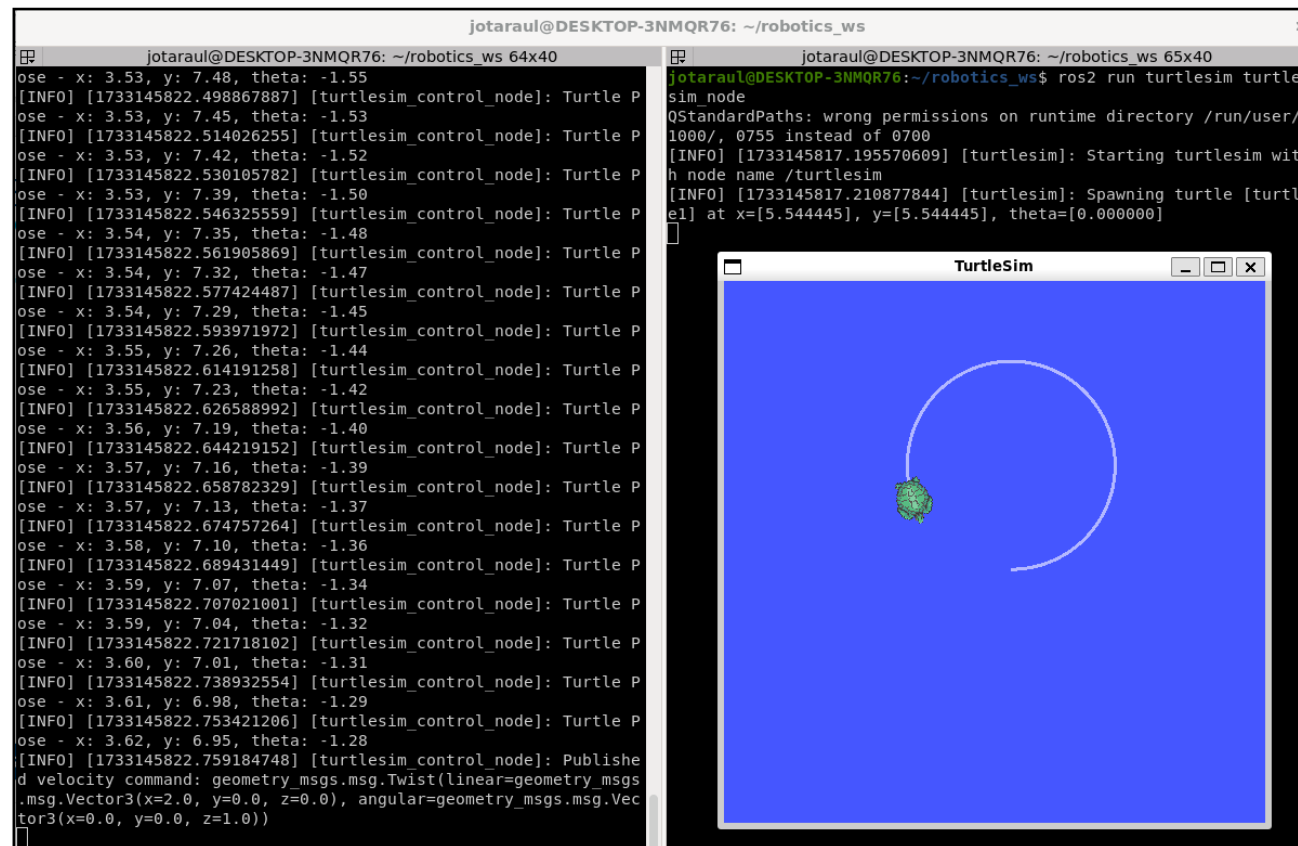
You may need to run `$ source ~/robotics_ws/install/setup.bash` after this to update all the dependencies chain.

## 2. ROS 2 – DEVELOPING SOFTWARE

### IMPLEMENTING YOUR OWN NODE (PYTHON)

6. Run your node! For that you must specify the package and node names:

```
$ ros2 run turtlesim_control turtlesim_control_node
```



The screenshot shows a terminal window with two panes. The left pane displays the output of the `ros2 run turtlesim_control turtlesim_control_node` command, showing a series of log messages from the `turtlesim_control_node` and `turtlesim` packages. The right pane shows the output of the `ros2 run turtlesim turtlesim_node` command, which includes a warning about permissions on the runtime directory and a message indicating that the `turtlesim` node is starting. Below the terminal output, a `TurtleSim` window is visible, showing a green turtle moving in a spiral pattern on a blue background.

```
jotaraul@DESKTOP-3NMQR76: ~/robotics_ws
jotaraul@DESKTOP-3NMQR76: ~/robotics_ws 64x40
ose - x: 3.53, y: 7.48, theta: -1.55
[INFO] [1733145822.498867887] [turtlesim_control_node]: Turtle P
ose - x: 3.53, y: 7.45, theta: -1.53
[INFO] [1733145822.514026255] [turtlesim_control_node]: Turtle P
ose - x: 3.53, y: 7.42, theta: -1.52
[INFO] [1733145822.530105782] [turtlesim_control_node]: Turtle P
ose - x: 3.53, y: 7.39, theta: -1.50
[INFO] [1733145822.546325559] [turtlesim_control_node]: Turtle P
ose - x: 3.54, y: 7.35, theta: -1.48
[INFO] [1733145822.561905869] [turtlesim_control_node]: Turtle P
ose - x: 3.54, y: 7.32, theta: -1.47
[INFO] [1733145822.577424487] [turtlesim_control_node]: Turtle P
ose - x: 3.54, y: 7.29, theta: -1.45
[INFO] [1733145822.593971972] [turtlesim_control_node]: Turtle P
ose - x: 3.55, y: 7.26, theta: -1.44
[INFO] [1733145822.614191258] [turtlesim_control_node]: Turtle P
ose - x: 3.55, y: 7.23, theta: -1.42
[INFO] [1733145822.626588992] [turtlesim_control_node]: Turtle P
ose - x: 3.56, y: 7.19, theta: -1.40
[INFO] [1733145822.644219152] [turtlesim_control_node]: Turtle P
ose - x: 3.57, y: 7.16, theta: -1.39
[INFO] [1733145822.658782329] [turtlesim_control_node]: Turtle P
ose - x: 3.57, y: 7.13, theta: -1.37
[INFO] [1733145822.674757264] [turtlesim_control_node]: Turtle P
ose - x: 3.58, y: 7.10, theta: -1.36
[INFO] [1733145822.689431449] [turtlesim_control_node]: Turtle P
ose - x: 3.59, y: 7.07, theta: -1.34
[INFO] [1733145822.707021001] [turtlesim_control_node]: Turtle P
ose - x: 3.59, y: 7.04, theta: -1.32
[INFO] [1733145822.721718102] [turtlesim_control_node]: Turtle P
ose - x: 3.60, y: 7.01, theta: -1.31
[INFO] [1733145822.738932554] [turtlesim_control_node]: Turtle P
ose - x: 3.61, y: 6.98, theta: -1.29
[INFO] [1733145822.753421206] [turtlesim_control_node]: Turtle P
ose - x: 3.62, y: 6.95, theta: -1.28
[INFO] [1733145822.759184748] [turtlesim_control_node]: Publishe
d velocity command: geometry_msgs.msg.Twist(linear=geometry_msgs
.msg.Vector3(x=2.0, y=0.0, z=0.0), angular=geometry_msgs.msg.Vec
tor3(x=0.0, y=0.0, z=1.0))
jotaraul@DESKTOP-3NMQR76: ~/robotics_ws 65x40
jotaraul@DESKTOP-3NMQR76: ~/robotics_ws$ ros2 run turtlesim turtle
sim_node
QStandardPaths: wrong permissions on runtime directory /run/user/
1000/, 0755 instead of 0700
[INFO] [1733145817.195570609] [turtlesim]: Starting turtlesim wit
h node name /turtlesim
[INFO] [1733145817.210877844] [turtlesim]: Spawning turtle [turtl
e1] at x=[5.544445], y=[5.544445], theta=[0.000000]
```

# REFERENCES

## WHERE TO FIND MORE KNOWLEDGE

- [1] ROS 2 Documentation: <https://docs.ros.org/en/humble/index.html>
- [2] Quigley, Morgan, et al. "ROS: an open-source Robot Operating System." *ICRA workshop on open source software*. Vol. 3. No. 3.2. 2009.
- [3] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, W. Woodall, "Robot Operating System 2: Design, architecture, and uses in the wild," *Science Robotics* vol. 7, May 2022.



SEE YOU IN THE NEXT LECTURE!