2.2 Environment

One highly important step is to always source the right ROS2 distribution when working with ROS2 in each tab of the terminal. This has been already covered in the previous practice.

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

If we don't want to add this step to our workflow every time we open a new tab or terminal window we can tell our terminal to automatically do this for us. For that, we need to update our /.bashrc file's content. We can do that with either a text editor inside Ubuntu like nano or just echo the correspondent command like so:

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

However, using this latter method is only good practice when using only one distribution of ROS2 at a time, but we shall see some advice making our life a little easier regarding the usage of multiple distributions at the same time at the end of this practice in the Cooking tips section 2.9.

From now we will not mention it, but in each case, a new tab will be needed to be opened You will need to source the ROS2 distribution if You choose not to echo the command to Your /.bashrc file, if You updated it then You are good to follow along the tutorials without any change.

Sometimes it is useful to check our environmental variables, for example, to check that we sourced correctly our ROS2 distribution. For this we can use the following command line:

```
$ printenv | grep -i ROS
```

In our current case we should get back an answer like this, with some extra info:

```
ROS_VERSION=2
ROS_PYTHON_VERSION=3
ROS_DISTRO=foxy
```

Next, we need to assign a domain ID for our ROS system. More detail about domain ID can be achieved from the concepts section of the documentation [1], but for now, it is enough to us that communicating nodes in the same ID group can freely find each other, and for Ubuntu is safe to choose a number between $0 - 101$ and $215 - 232$. To assign this domain ID we will need to update our /.bashrc file, for which we will use the echo command again:

```
$ echo "export ROS_DOMAIN_ID=1" >> ~/.bashrc
```

Lastly, for our learning purposes, we need to limit the communication to the local host. With this, our communication between nodes becomes invisible on the local network (which can save us some headaches with the strange behaviors, caused by the multiple occupation of the same communication channel). For this, we update the /.bashrc file:

```
$ echo "export ROS_LOCALHOST_ONLY=1" >> ~/.bashrc
```

[1] https://docs.ros.org/en/foxy/Concepts/About-Domain-ID.html

2.3 Turtlesim with ros2

After we configured our system we can start familiarizing ourselves with it. For this, we will first install and use a package called Turtlesim. Let's install it (don't forget to check the system's up-to-dateness):

```
$ sudo apt update
$ sudo apt install ros-foxy-turtlesim
```

To further check the installation's success we can use the following command:

```
$ ros2 pkg executables turtlesim
```

This will give us back the following response if the installation was sucessfull:

```
turtlesim draw_square
turtlesim mimic
turtlesim turtle_teleop_key
turtlesim turtlesim_node
```

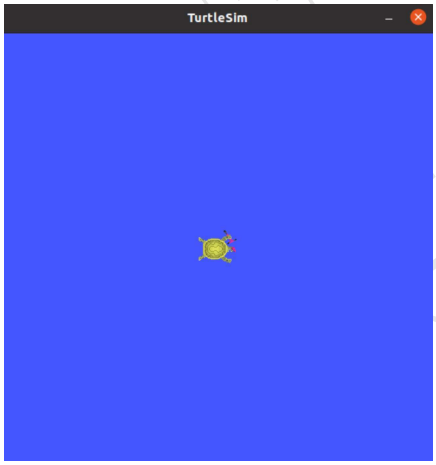
Now we can start our nodes of turtlesim package. To run a node in ROS2 we can use the 'run command', which has the following sintax:

```
$ ros2 run <PACKAGE_NAME> <NODE_NAME> <PARAMETERS>
```

This said we start the turtlesim package's turtlesim_node node.

```
$ ros2 run turtlesim turtlesim_node
```

A new window, similar to the figure below, should appear with a random turtle in its center.



We should also get some response in the terminal window as a feedback:

```
[INFO] [turtlesim]: Starting turtlesim with node name /turtlesim
[INFO] [turtlesim]: Spawning turtle [turtle1] at x=[5.544445], y
   =[5.544445], theta=[0.000000]
```

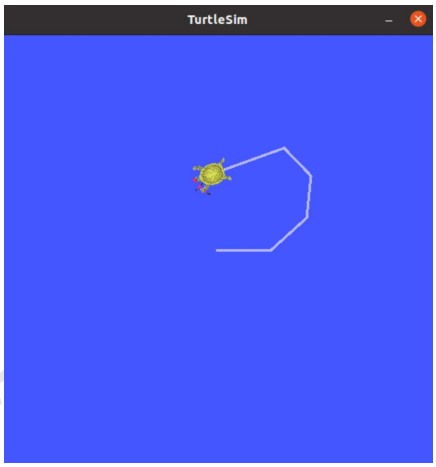
This simply states that our turtle's name is turtlesim (since we did not give any name), and it was spawned with the x, y, and theta coordinates.

So far this is good, but not happening any interesting. No movement, we can't communicate with the turtle. We can command the turtlesim relatively easily, for which we

will run another node called 'turtle_teleop_key'. This will allow us to maneuver the turtle with the arrow buttons. Open a new terminal and in that terminal :

```
$ ros2 run turtlesim turtle_teleop_key
```

Now we can control the turtle with the keyboard:



2.4 Nodes

We already know how to run a node, but we can also check all nodes currently running in our system. For that, we will use the following command in a new tab of the terminal:

```
$ ros2 node list
```

Since our turtlesim node and teleop key nodes are still running, it will not be a big surprise, that we get the following response:

```
/teleop_turtle
/turtlesim
```

If we want to get more information about the nodes that are currently running we can use the 'info' command (on the turtlesim node for example) as follows:

```
$ ros2 node info /turtlesim
```

And we will get back the following information, where we can see all the possible communication done by the node. It is organized according to the communication protocols available in ROS2. Each line consists of two parts, the name of the communication channel and the type of information used in that channel.

```
/turtlesim
  Subscribers:
    /parameter_events: rcl_interfaces/msg/ParameterEvent
    /turtle1/cmd_vel: geometry_msgs/msg/Twist
  Publishers:
    /parameter_events: rcl_interfaces/msg/ParameterEvent
    /rosout: rcl_interfaces/msg/Log
    /turtle1/color_sensor: turtlesim/msg/Color
```

```
    /turtle1/pose: turtlesim/msg/Pose
  Service Servers:
    /clear: std_srvs/srv/Empty
    /kill: turtlesim/srv/Kill
    /reset: std_srvs/srv/Empty
    /spawn: turtlesim/srv/Spawn
    /turtle1/set_pen: turtlesim/srv/SetPen
    /turtle1/teleport_absolute: turtlesim/srv/TeleportAbsolute
    /turtle1/teleport_relative: turtlesim/srv/TeleportRelative
    /turtlesim/describe_parameters: rcl_interfaces/srv/
   DescribeParameters
    /turtlesim/get_parameter_types: rcl_interfaces/srv/GetParameterTypes
    /turtlesim/get_parameters: rcl_interfaces/srv/GetParameters
    /turtlesim/list_parameters: rcl_interfaces/srv/ListParameters
    /turtlesim/set_parameters: rcl_interfaces/srv/SetParameters
    /turtlesim/set_parameters_atomically: rcl_interfaces/srv/
   SetParametersAtomically
  Service Clients:

  Action Servers:
    /turtle1/rotate_absolute: turtlesim/action/RotateAbsolute
  Action Clients:
```

It is possible to rename the basic parameters of a node, which is called remapping in ROS2. This can come in handy if we use multiple instances of the same node. Let's run a new instance of the turtlesim, but now we give a name to our turtle. We will run our node as usually in the third tab of the terminal, but now we add extra parameters to it '–ros-args –remap __node:=my_turtle' that will tell the system that now we want to rename our instance to 'my_turtle':

```
$ ros2 run turtlesim turtlesim_node --ros-args --remap __node:=my_turtle
```

Now checking with 'node list' we will get the following answer:

```
/my_turtle
/teleop_turtle
/turtlesim
```

And also we will now see two windows with turtles:

2.5 Parameters

In ROS2 parameters are the node's private configuration values. We can get, set, save, and reload them if needed.

We can list all the currently running node's parameters with:

```
$ ros2 param list
```

That gives us the following info about our three running nodes. These are the parameters used when the node is started. we can see background colors RGB parameters and sim time for the turtlesims:

```
/my_turtle:
  background_b
  background_g
  background_r
  use_sim_time
/teleop_turtle:
  scale_angular
  scale_linear
  use_sim_time
/turtlesim:
  background_b
  background_g
  background_r
  use_sim_time
```

To get the value of a given parameter we need to use the 'get' command as follows:

```
$ ros2 param get /my_turtle background_g
```

And we get the answer:

```
Integer value is: 86
```

Let's set this to another value and see what happens. We can do so with the 'set' command:

```
ros2 param set /turtlesim background_g 150
```

We get the following response:

```
Set parameter successful
```

And our background for 'my_turtle" (right) has changed:

Figure 2.1: Caption

If this color is really for Your liking, then You can save it with the following command:

```
ros2 param dump /my_turtle
```

This will save the current parameter values to the file in the response:

```
Saving to:  ./my_turtle.yaml
```

We can load this parameter set in two different ways. While running the node or at the start of the node's execution. To try it out first terminate the execution of the 'my_turtle', then rerun it similarly as we did it earlier.

For the prior, we will use the following command (to recolor the background of our turtlesim):

```
$ ros2 param load /my_turtle ./my_turtle.yaml
```

And for the latter, we again terminate the execution, and then we will use the following extension od the 'ros2 run' command

```
$ ros2 run turtlesim turtlesim_node --ros-args --remap __node:=my_turtle
    --params-file ./my_turtle.yaml
```

Which starts a node with the exact background color we wanted.

2.6 Launch - basics

We will delve deeper into the topic of launch files later, but it is a wise strategy to have at least some minimum understanding of the launch files at this stage. We can use two methods to start a node, the first one is the 'run' command, which we already covered. The

second one comes in handy when we want to start multiple nodes with special attributes at the start. This method can be done by launch files. To see this in work, let's now close all nodes that we currently have and in a blank tab we use the following command:

```
$ ros2 launch turtlesim multisim.launch.py
```

This command tells ROS2 that we want to execute a launch file, called multisim.launch.py. The syntax of the 'launch' command is pretty similar compared to the 'run' command (launch + package + launchfilename).

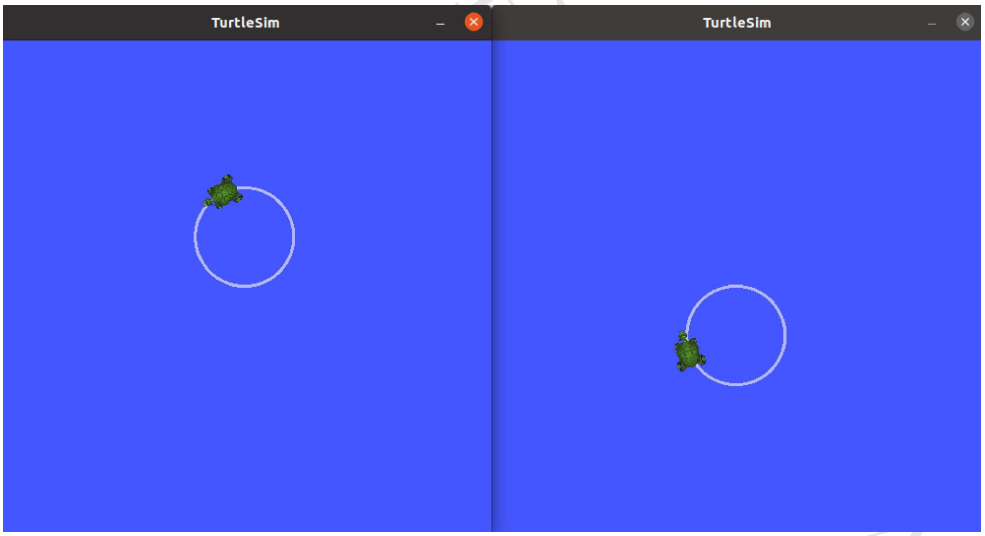
We can see that now two turtlesim node and window has started simultaneously. After checking on the current running nodes with the node list we can see that the names of the turtlesim nodes are: 'turtlesim1' and 'turtlesim2'

We can try to command them with the following two commands (in two new tabs):

```
$ ros2 topic pub  /turtlesim1/turtle1/cmd_vel geometry_msgs/msg/Twist "{
    linear: {x: 2.0, y: 0.0, z: 0.0}, angular: {x: 0.0, y: 0.0, z: 1.8}}"
```

```
$ ros2 topic pub  /turtlesim2/turtle1/cmd_vel geometry_msgs/msg/Twist "{
    linear: {x: 2.0, y: 0.0, z: 0.0}, angular: {x: 0.0, y: 0.0, z: -1.8}}
    "
```

The first will make the turtlesim1's turtle move in a circle counter-clockwise, while the second command will tell the turtlesim2's turtle to move in a circle clockwise. Don't care now to understand what 'topic' and 'pub' and all the other stuff means, we will cover it later. For now, it is only important, that we have two turtles with two communication channels and we publish our commands at a certain frequency for continuous movement.



So now, we can see the light at the end of the dungeons full of new terminal tabs. Unfortunately, most of the time we will use multiple terminal tabs, but with launch files this number can be reduced hugely in projects with big numbers of nodes working together.

2.7 RQT - How to observe the connections between nodes

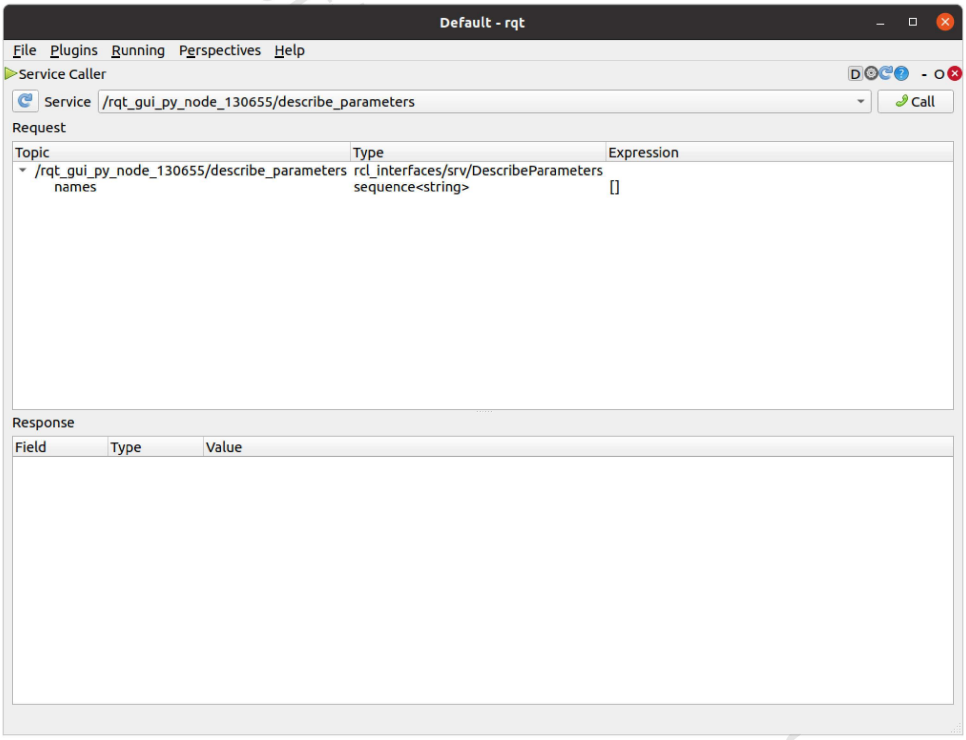
One of the most important packages of ROS2, when debugging and understanding systems is the RQT package, so let's install it:

```
$ sudo apt update
$ sudo apt install ~nros-foxy-rqt*
```

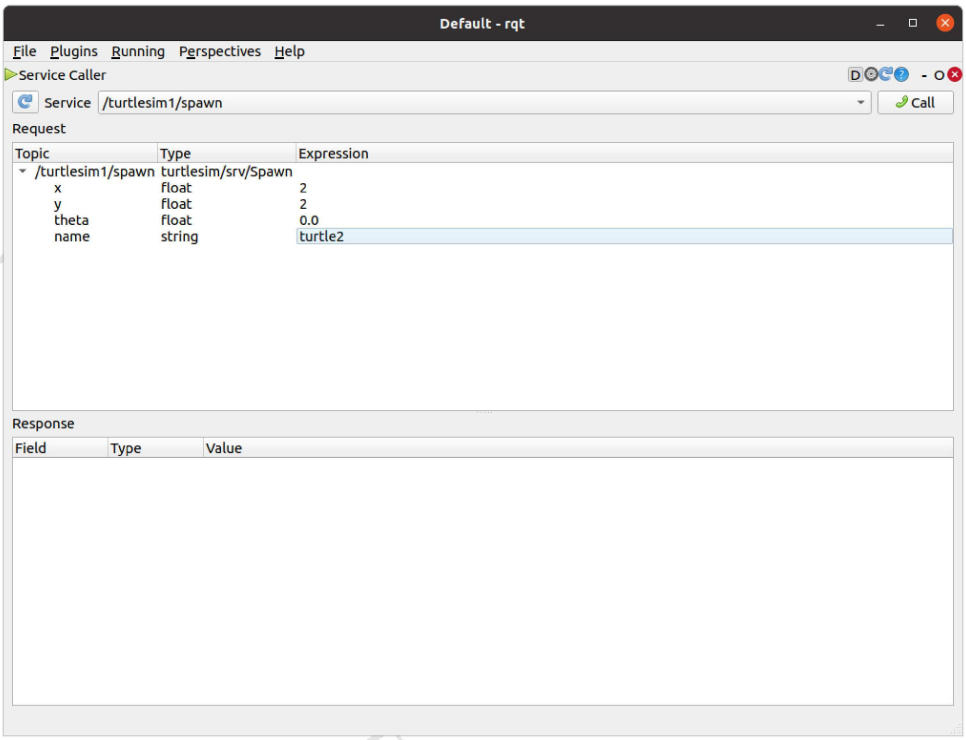
RQT has a bunch of important features for us. We can see the communication between nodes, the connections can be visualized, send commands, set parameters etc. First, we will round rqt.

```
$ rqt
```

When calling 'rqt' the first time, we will face a blank window. We will try to spawn a new turtle for one of our turtlesims. For this, we will use something called service. So click on plugins/Services/Service caller. We will see the following window:



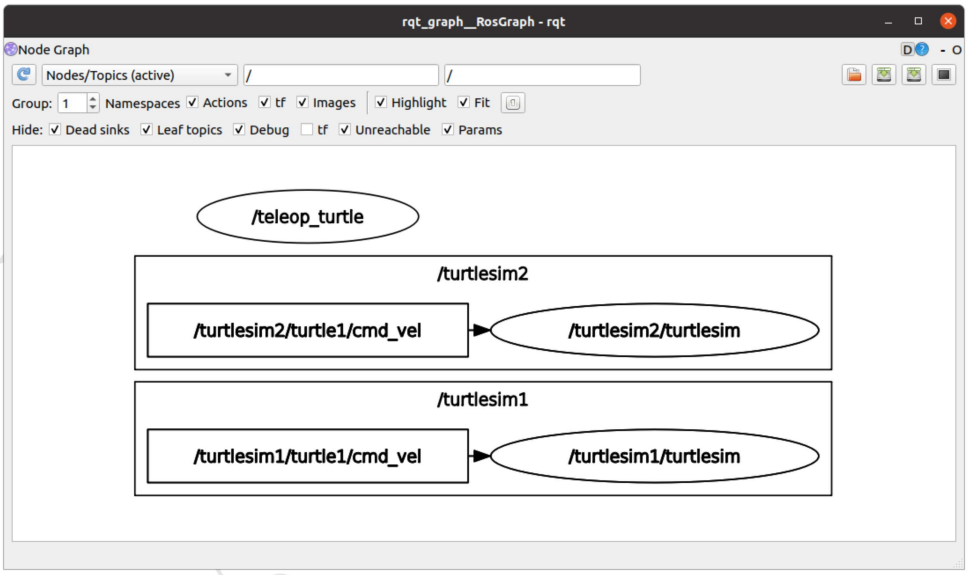
To spawn a new turtle we will use the '/spawn' service. Let's find the turtlesim1's spawn service (/turtlesim1/spawn) in the upper list. When selected, we will see the following window, where we can select the coordinates of the new spawned turtle. We can easily change that by clicking on the corresponding field. Also, we need to give our new turtle a name, this is the time to be creative, so we will name it turtle2:



Clicking on the call button we can see the new turtle spawned successfully.

We can also see how the currently running nodes connect to each other. for that we restart the 'teleop_key' node and then start the 'rqt_graph' in a separate tab.

We can see that in this graph there are three nodes (encircled), and we currently have two active communication channels (in rectangles). we can also see that there is no communication between the three nodes since there is no line connecting them via a communication channel. Indeed, if we try to use the 'teleop_key' we will see that the turtles now are not moving. In a real-life scenario, this can be a really good hint when we troubleshoot or debug our system.

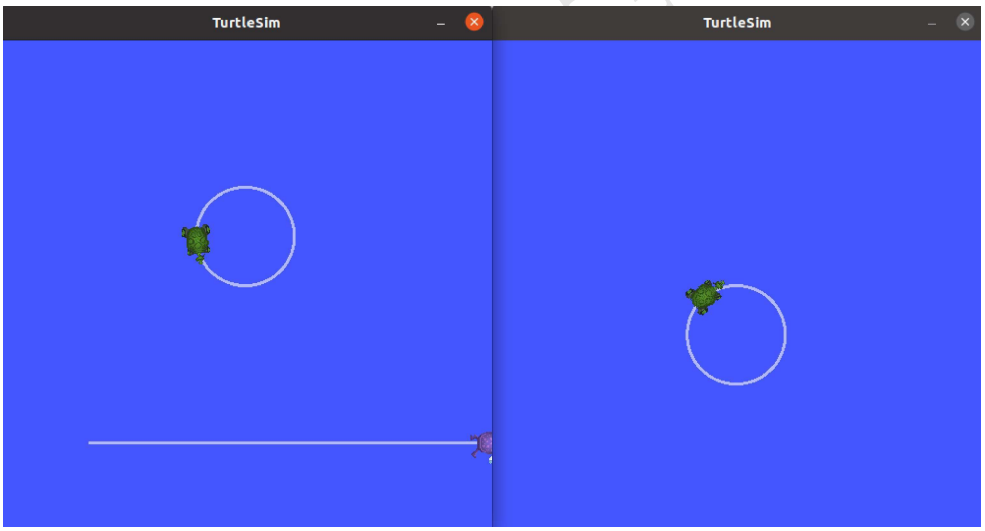


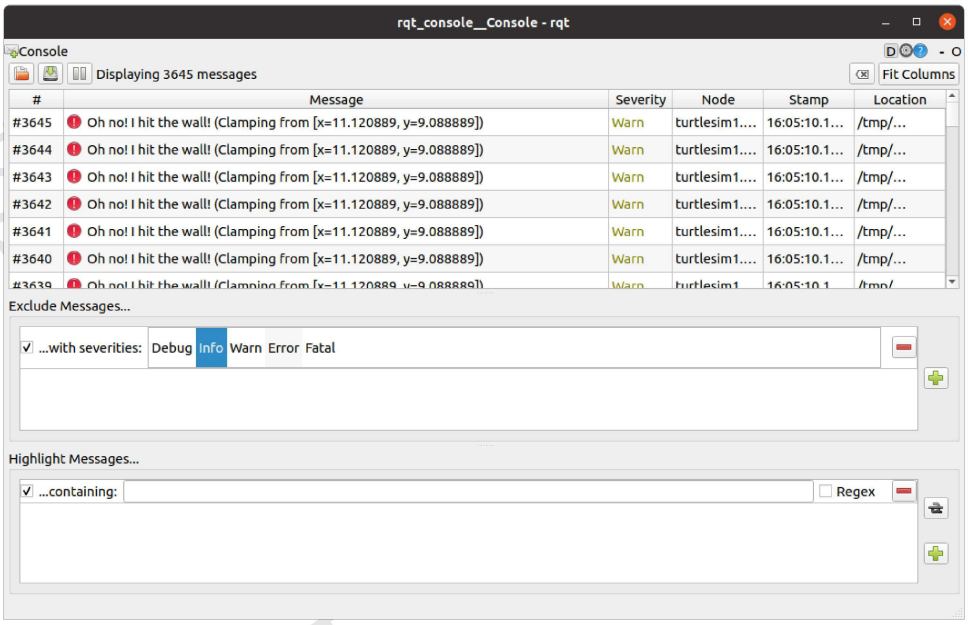
The third important tool inside rqt is the 'rqt_console', which allows us to view the logs from the nodes in situ. To start 'rqt_console' we will use the following command:

```
$ ros2 run rqt_console rqt_console
```

Since currently nothing bad is happening with our turtles we can't see any messages. Let's shake things up a little and send a message to turtel2 so it will collide with the side of the turtlesim window.

```
$ ros2 topic pub -r 1 turtlesim1/turtle2/cmd_vel geometry_msgs/msg/Twist
    "{linear: {x: 2.0, y: 0.0, z: 0.0}, angular: {x: 0.0,y: 0.0,z: 0.0}}
  "
```





And there it is. Our turtle2 says: "Oh no! I hit the wall....."

2.8 Bags - Recording and Playback of a session

Sometimes we need to record our sessions for later analysis, for this ROS2 gives us help in the form of bag files. To make our learning easier, now stop all running nodes and clear all tabs, we will not need those for now. We will control our turtle, record the session and then replay it. For that, in two new tabs we run 'turtlesim' and 'teleop_key':

```
$ ros2 run turtlesim turtlesim_node
```

```
$ ros2 run turtlesim turtle_teleop_key
```

In a third tab we will make our folder for the bag files and also select it as a working directory.

```
$ mkdir bag_files
$ cd bag_files
```

To record the movement of the turtle we will need to save its '/turtle1/cmd_vel' channels content. After this, we can make a short play with our turtle, and with "ctrl+C" (in the recordings window) we can stop the recording.

```
$ ros2 bag record -o firstrecord /turtle1/cmd_vel
```

We named our recording "firstrecord" with the help of the "-o" argument. Let's check it's details now:

```
$ ros2 bag info firstrecord
```

We will get a response like the following:

```
Files:             firstrecord_0.db3
Bag size:          16.8 KiB
```

```
Storage id:          sqlite3
Duration:            6.898s
Start:               Aug  3 2024 16:26:27.669 (1722695187.669)
End:                 Aug  3 2024 16:26:34.567 (1722695194.567)
Messages:            16
Topic information: Topic: /turtle1/cmd_vel | Type: geometry_msgs/msg/
   Twist | Count: 16 | Serialization Format: cdr
```

Finally, let's replay our session:

```
$ ros2 bag play firstrecord
```

There are more options for the recording, like recording multiple or all possible communications etc. Usually, it depends on the task and what we want to record for the later analysis or demo.

This concludes our second practice. I hope You had fun and learned a lot. We covered a lot of different topics, but this is just the way how all starting is, hard. In the next sessions, we will start to focus on more narrow topics, so if You feel a little overwhelmed, it is okay, and don't worry, we will learn more complex but less divers things from now.

2.9 Cooking tips

2.9.1 Sourcing with multiple distributions

Alternatively, when using different distributions, we can use aliases for easy access to this command. For this, we need to update our /.bashrc file with the new aliases (with a text editor or the echo command) for each distribution that we want to access easily. For example, we can have something similar convention inside the /.bashrc like the following:

```
alias sourcefoxy='source /opt/ros/foxy/setup.bash'
```