ROS 2 from scratch – 2024

*Pack Publishing*



*By Edward Renard*

Note: I have a Udemy course with basically the same info.

# Part 1: Getting Started with ROS 2

# Chapter 1 - Introduction to ROS 2 – What Is ROS 2?

## 1.1 Terminology

## 1.2 What is ROS, when should we use it, and why?

### Why ROS?

### What is ROS?

ROS is a combination of four main parts:

* Framework
* Set of tools
* Plug-and-play plugins

**Online community**

You can find all the ROS code online, as well as the code for the plug-and-play plugins. Everything is easily accessible on GitHub.

The ROS project is also backed by an online community that you can most commonly find in the following areas:

* **Robotics Stack Exchange** (<https://robotics.stackexchange.com/>): You can use this to ask technical questions. If you know Stack Overflow, as most developers do, well, this is Stack Overflow for robotics.
* **ROS Discourse forums** (<https://discourse.ros.org/>): Here, you can get informed about the latest developments, jobs, community projects, new ideas, and more. I recommend checking this website often to stay up to date with where ROS is going.

### When to use ROS

An ***actuator*** is something that creates movement (for example, a motor to rotate a wheel).

## 1.3 ROS 1 versus ROS 2

A quick story of ROS, and how we got to ROS 2

Is ROS 1 dead already?

## 1.4 Prerequisites for starting with ROS 2

Knowledge prerequisites

Hardware and software

## 1.5 How to follow this book

# Chapter 2 - Installing and Setting Up ROS 2

## 2.1 Which ROS 2 distribution to choose

### What is a ROS 2 distribution?

Every year, a new ROS 2 distribution is released on May 23. This day corresponds to *World Turtle Day*.

In May 2024, ROS ***Jazzy Jalisco*** was released.

### LTS and non-LTS distributions

### How to choose a ROS distribution

## 2.2 Installing the OS (Ubuntu)

### The relationship between ROS 2 and Ubuntu

### Installing Ubuntu 24.04 natively with a dual boot

### Installing Ubuntu 24.04 on a VM

## 2.3 Installing ROS 2 on Ubuntu

### Setting the locale

### Setting up the sources

$ sudo apt install software-properties-common

$ sudo add-apt-repository universe

$ sudo apt update && sudo apt install curl -y

$ sudo curl -sSL https://raw.githubusercontent.com/ros/rosdistro/master/ros.key -o /usr/share/keyrings/ros-archive-keyring.gpg

$ echo "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/ros-archive-keyring.gpg] http://packages.ros.org/ros2/ubuntu $(. /etc/os-release && echo $UBUNTU\_CODENAME) main" | sudo tee /etc/apt/sources.list.d/ros2.list > /dev/null

The main goal of these commands is to add the ROS packages server to your apt sources. Run the following command:

$ sudo apt update

### Installing ROS 2 packages

Install ROS Desktop by running:

$ sudo apt install ros-jazzy-desktop

install the ROS development tools:

$ sudo apt install ros-dev-tools

## 2.4 Setting up the environment for ROS 2

### Sourcing ROS 2 in the environment

source this bash script from where ROS 2 is installed:

$ source /opt/ros/jazzy/setup.bash

### Adding the source line to the .bashrc file

You must source this bash script every time you open a new session or Terminal. To make things easier and so you don’t forget about it, let’s just add this command line to the .bashrc file.

If you don’t know what .bashrc is, simply put, it’s a bash script that will run every time you open a new session (that can be an SSH session, a new Terminal window, and so on). This .bashrc file is specific to each user, so you will find it in your home directory (as a hidden file because of the leading dot).

You can add the source line to the .bashrc file with this command:

$ echo 'source /opt/ros/jazzy/setup.bash' >> ~/.bashrc

Now, to make a final check, open a Terminal and run this command (there’s no need to understand anything for now; it’s just to verify the installation):

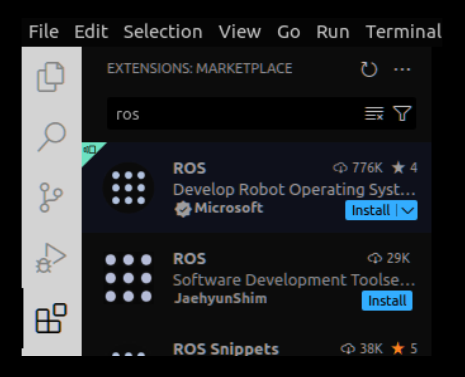
$ ros2 run turtlesim turtlesim\_node

## 2.5 Extra tools for ROS development 29

### Visual Studio Code

Now, start VS Code and go to the *Extensions* panel—you can find it on the left menu.

There, you can search for the ROS extension by typing ros. There are quite a few; choose the one developed by ***Microsoft***. This extension is compatible with both ROS 1 and ROS 2, so there’s no problem here:



Install this extension. This will also install a bunch of other extensions, notably Python and C++ extensions, which are quite useful when writing code.

On top of this, I also usually install the ***CMake*** extension by twxs (just type cmake and you’ll find it). With this, we get nice syntax highlighting when writing into **CMakeLists.txt** files, which is something we will have to do quite often with ROS 2.

### The Terminal and other tools

As you develop with ROS 2, you will often need to open several Terminals: one for compiling and installing, a few to run the different programs of your application, and a few more for introspection and debugging.

It can become quite difficult to keep track of all the Terminals you use, so as best practice, it’s nice to have a tool that can easily handle multiple Terminals in one window.

There are quite a few tools for doing this. The one I’m going to talk about here is called ***terminator***. Not only does it have a funny name, but it’s also super practical to use.

To install terminator, run the following command:

$ sudo apt install terminator

Then, you can find it from the applications menu, run it, right-click on the left bar menu, and choose Pin to Dash so that it stays there and becomes easy to start.

Here are the most important terminator commands:

* *Ctrl + Shift + O*: split the selected Terminal horizontally.
* *Ctrl + Shift + E*: split the selected Terminal vertically.
* *Ctrl + Shift + X*: make the current Terminal fill the entire window. Use again to revert.
* *Ctrl + Shift + W*: close a Terminal.

## 2.6 Summary

# Chapter 3 - Uncovering ROS 2 Core Concepts

a ROS 2 program is called a ***node***.

## 3.1 Running your first node

### Starting a node from the terminal with ros2 run

To start a node, you have to follow this template:

ros2 run <package> <executable>

As we will see later, nodes are organized inside packages. That’s why you first need to specify the package name where the node is and the executable name for that node.

In Terminal 1, start the talker node from the demo\_nodes\_cpp package:

$ ros2 run demo\_nodes\_cpp talker

In another terminal, let’s start a different node, which is the listener node from the same package:

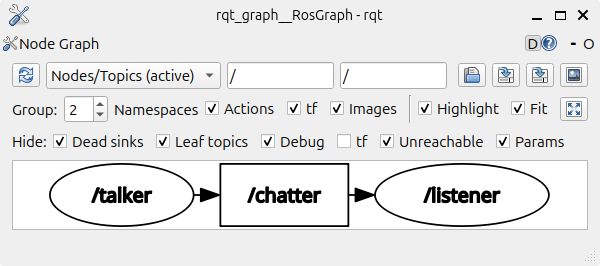
$ ros2 run demo\_nodes\_cpp listener

### Introspecting the nodes with rqt\_graph

There is another very useful tool we will discover here, which is a good complement to the command line: ***rqt\_graph***. This tool will show you all running nodes with a nice visual.

$ rqt\_graph

This will open a new graphical window, where you should see the two nodes.



### Running a 2D robot simulation

In Terminal 1, run the following command:

$ ros2 run turtlesim turtlesim\_node

In Terminal 2, start this second node:

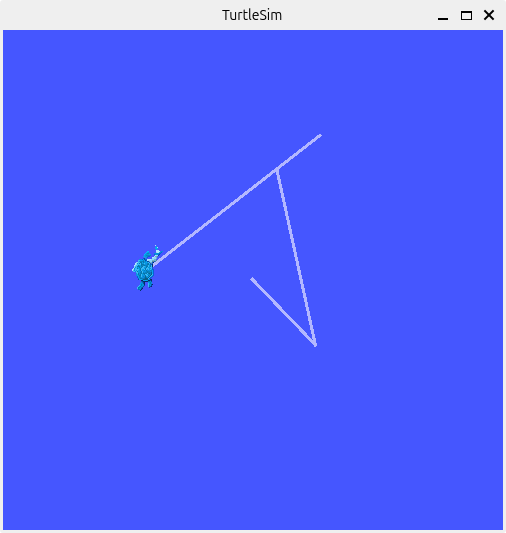
$ ros2 run turtlesim turtle\_teleop\_key

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.



### Recap – nodes

## 3.2 Topics

Nodes communicate with each other using ROS 2 communication features. There are three types of communication: *topics*, *services*, and *actions*.

### Running a topic publisher and subscriber

In the middle, you will see a /chatter box. This box represents a ROS 2 *topic*. What you can also see is that the talker node is sending something to the /chatter topic, which will then be received by the listener node.

We say that the talker is a ***publisher***, and the listener is a ***subscriber***.

### A name and an interface (data type)

The ros2 topic list command will list all available topics, which means all topic communications between running nodes.

$ ros2 topic list

What kind of data is being sent?

To get this information, we can use ros2 topic info <topic\_name>:

$ ros2 topic info /chatter

Type: std\_msgs/msg/String

Publisher count: 1

Subscription count: 1

Here, we see how many nodes are publishing and subscribing to this topic. We have one publisher (talker node) and one subscriber (listener node). We can also see what kind of message is being sent: std\_msgs/msg/String. In ROS 2, this message is called an ***interface***.

To see what’s inside an interface, run ros2 interface show <interface\_name>:

$ ros2 interface show std\_msgs/msg/String

# Some comments

string data

### More experimentation with topics

### Recap – topics

## 3.3 Services

Topics are very useful to send a stream of data/commands from one node to another node. However, this is not the only way to communicate. You can also find client/server communications in ROS 2. In this case, ***services*** will be used.

### Running a service server and client

We will also start a client node, which will send a request to the server node.

In Terminal 1, input the following:

$ ros2 run demo\_nodes\_cpp add\_two\_ints\_server

In Terminal 2, input the following:

$ ros2 run demo\_nodes\_cpp add\_two\_ints\_client

### A name and an interface (data type)

As for topics, services are defined by two things: a name, and an interface (data type). The only difference is that the interface will contain two parts: a request and a response.

To find the name of the service, we can use the ros2 command-line tool again, this time with the ***service*** command, followed by ***list***.

$ ros2 service list

Now, to get the data type, we can use ros2 service type , and then ros2 interface show :

$ ros2 service type /add\_two\_ints

example\_interfaces/srv/AddTwoInts

$ ros2 interface show example\_interfaces/srv/AddTwoInts

int64 a

int64 b

---

int64 sum

### Sending a request from the terminal

Instead of running the add\_two\_ints\_client node, we can also send a request directly from the terminal. This is a very useful way to test a service without requiring an existing client node.

The syntax is ros2 service call <service\_name> <interface\_name> “<request\_in\_json>”.

Example:

$ ros2 service call /add\_two\_ints example\_interfaces/srv/AddTwoInts "{a: 4, b: 7}"

waiting for service to become available...

requester: making request: example\_interfaces.srv.AddTwoInts\_Request(a=4, b=7)

response:

example\_interfaces.srv.AddTwoInts\_Response(sum=11)

### More experimentation with services

### Recap – services

## 3.4 Actions

A ROS 2 action is basically the same thing as a service (client/server communication), but designed for longer tasks, and when you might want to also get some feedback during the execution, be able to cancel the execution, and so on.

### Running an action server

$ ros2 run turtlesim turtlesim\_node

$ ros2 action list

/turtle1/rotate\_absolute

From what we observe, it seems that the turtlesim node contains an action server named /turtle1/rotate\_absolute.

### A name and an interface (data type)

For actions, the interface contains three parts: **goal**, **result**, and ***feedback***.

The goal and result are similar to the request and response for a service. The feedback is additional data that can be sent by the server to give some feedback during the goal execution.

To get the action interface, you can run the ros2 action info <action\_name> -t command.

$ ros2 action info /turtle1/rotate\_absolute -t

Action: /turtle1/rotate\_absolute

Action clients: 0

Action servers: 1

    /turtlesim [turtlesim/action/RotateAbsolute]

We can see that the action is running within one server (the turtlesim node), and we also found the interface: turtlesim/action/RotateAbsolute.

### Sending a goal from the terminal

The syntax to send a goal from the terminal is ros2 action send\_goal <action\_name> <action\_interface> “goal\_in\_json>”.

send a goal from Terminal 2:

$ ros2 action send\_goal /turtle1/rotate\_absolute turtlesim/action/RotateAbsolute "{theta: 1.0}"

Waiting for an action server to become available...

Sending goal:

theta: 1.0

Goal accepted with ID: 3ba92096282a4053b552a161292afc8e

Result:

delta: -0.9919999837875366

Goal finished with status: SUCCEEDED

After you run the command, you should see the turtle robot rotate on the 2D window. Once the desired angle is reached, the action will finish, and you will receive the result.

### Recap – actions

## 3.5 Parameters

***parameters*** are used to give different settings to a node when you start it.

### Getting the parameters for a node

to list all parameters:

$ ros2 param list

To get the value of a parameter we use ros2 param get <node\_name> <param\_name> :

$ ros2 param get /turtlesim background\_b

Integer value is: 255

### Setting up a parameter value for a node

let’s modify the value ourselves when we start the node. For this, we will need to restart the node, using the same syntax as before: ros2 run <pacakge\_name> <executable\_name>. We will then add --ros-args (only once), and –p <param\_name>:=value for each parameter we want to modify.

$ ros2 run turtlesim turtlesim\_node --ros-args -p background\_b:=0 -p background\_r:=0

### Recap – parameters

## 3.6 Launch files

A **launch file** will allow you to start several nodes and parameters from just one file, which means that you can start your entire application with just one command line.

### Starting a launch file

For launch files, we will use ros2 launch .

$ ros2 launch demo\_nodes\_cpp talker\_listener\_launch.py

[INFO] [talker-1]: process started with pid [2868]

[INFO] [listener-2]: process started with pid [2871]

### Recap – launch files

## 3.7 Summary

# Part 2: Developing with ROS 2 – Python and C++

# Chapter 4. Writing and Building a ROS 2 Node

## 4.1 Technical requirements

## 4.2 Creating and setting up a ROS 2 workspace

### Creating a workspace

To create a workspace, you will simply create a new directory inside your home directory.

Let's call it *ros2\_ws* and let's create a "*scr*" directory inside it.

$ cd

$ mkdir ros2\_ws

$ cd ros2\_ws/

$ mkdir src

### Building the workspace

Let’s build the workspace:

$ cd ~/ros2\_ws/

$ colcon build

Summary: 0 packages finished [0.73s]

As you can see, no packages were built, but let’s list all directories under ~/ros2\_ws:

$ ls

build  install  log  src

As you can see, we have three new directories: build, install, and log. The build directory will contain the intermediate files required for the overall build. In log, you will find logs for each build. The most important directory for you is install, which is where all your nodes will be installed after you build the workspace.

### Sourcing the workspace

If you navigate inside the newly created install directory, you can see a setup.bash file.

To source the workspace, source this setup.bash script:

$ source ~/ros2\_ws/install/setup.bash

Then, as we previously did, we are going to add that line into our .bashrc. This way, you don’t need to source the workspace every time you open a new terminal.

The new "source" command must be put after the other in the ~/.bashrc file:

source/opt/ros/jazzy/setup.bash

source ~/ros2\_ws/install/setup.bash

Note: If you build the workspace in an already sourced environment, you will still need to source the workspace once again as there have been some changes, and the environment is not aware of that. In this case, you can either source the workspace’s setup.bash script directly, source the .bashrc, or open a new terminal.

### Sourcing a workspace according to Gemini:

In ROS 2, "sourcing the workspace" refers to the process of loading the necessary environment variables and libraries from your ROS 2 workspace into your current terminal session.

**Here's a breakdown:**

* **Workspace**: In ROS 2, a workspace is a directory where you develop and build your ROS 2 packages. It typically contains source code, build files, and installation directories.
* **Sourcing**: The act of sourcing a workspace involves executing a script (usually setup.bash or setup.sh) that:
* **Sets environment variables**: This includes paths to libraries, executables, and other necessary files within your workspace.
* **Loads libraries**: It makes the libraries built within your workspace available for use in your current terminal session.

**Why is it important?**

* **Access to workspace components**: By sourcing your workspace, you ensure that your terminal can find and use the packages, nodes, and other components you've developed within that workspace.
* **Running nodes**: To run nodes from your workspace, you must first source the workspace.
* **Using command-line tools**: Many ROS 2 command-line tools (like ros2 topic list, ros2 node info, etc.) rely on the environment variables set by sourcing the workspace.

## 4.3 Creating a package

### What is a ROS 2 package?

A ROS 2 package is a sub-part of your application.

Let’s consider a robotic arm that we want to use to pick up and place objects. Before we create any node, we can try to split this application into several sub-parts, or packages.

We could have **one package to handle a camera**, **another package for the hardware control (motors)**, and yet another package to compute motion planning for the robot.

**Each package is an independent unit, responsible for one sub-part of your application.**

### Creating a Python package

Command-line to create a package:

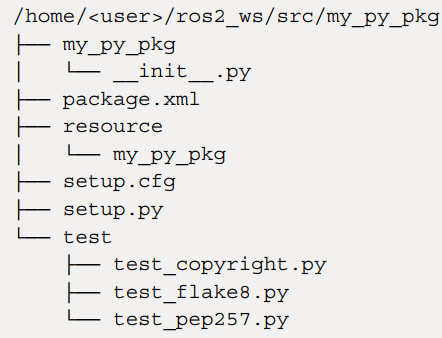
1. ros2 **pkg create** <pkg\_name>: This is the minimum you need to write.
2. You can specify a build type with **--build\_type** < build\_type>. For a Python package, we need to use **ament\_python**.
3. You can also specify some optional dependencies with **--dependencies** <list\_of\_ dependencieEg.s\_separated\_with\_spaces>. It’s always possible to add dependencies later in the package.

Eg.

$ ros2 pkg create my\_py\_pkg --build-type ament\_python --dependencies rclpy

With this command, we say that we want to create a package named *my\_py\_pkg*, with the **ament\_python** build type, and we specify one dependency: **rclpy**—this is the Python library for ROS 2 that you will use in every Python node.

You can then see that there is a new directory named my\_py\_pkg. Here is the architecture of your newly created Python package:



Here is a quick overview of the most important files and directories:

my\_py\_pkg: As you can see, inside the package, there is another directory with the same name. This directory already contains an \_\_init\_\_.py file. This is where we will create our Python nodes.

* package.xml: Every ROS 2 package (Python or C++) must contain this file. We will use it to provide more information about the package as well as dependencies.
* setup.py: This is where you will write the instructions to build and install your Python nodes.

### Creating a C++ package

### Building a package

To build the packages, go back to the root of your ROS 2 workspace and run colcon build. Once again, and as seen previously in this chapter, where you run this command is very important.

$ cd ~/ros2\_ws/

$ colcon build

Note: After you build any package, you also have to source your workspace so that the environment is aware of the new changes.

To build only a specific package, you can use the --packages-select option, followed by the name of the package. Here’s an example:

$ colcon build --packages-select my\_py\_pkg

### How are nodes organized in a package?

Nodes communicate with each other using ROS 2 communications (topics, services, and actions).

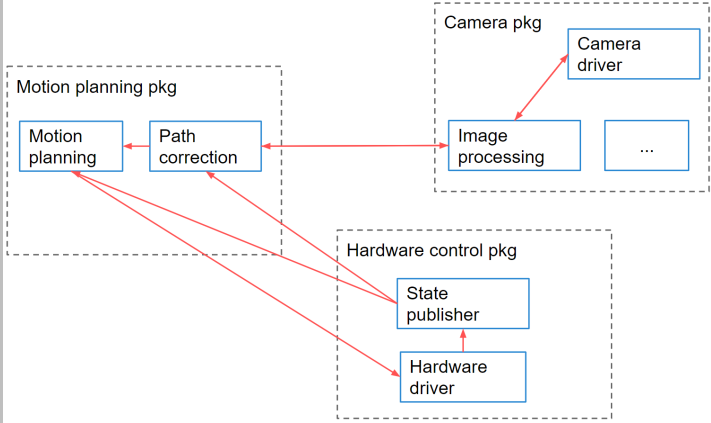


Figure 4.2 – Example of a package organization with nodes

## 4.4 Creating a Python node

### Creating a file for the node

If you remember, when we created the *my\_py\_pkg* package, another *my\_py\_pkg* directory was created inside the package. This is where we will write the node.

Create a new file in this directory and make it executable:

$ cd ~/ros2\_ws/src/my\_py\_pkg/my\_py\_pkg/

$ touch my\_first\_node.py

$ chmod +x my\_first\_node.py

Note: If you are using VS Code, the best way to open it is to first navigate to the src directory of your workspace in a terminal, and then open it. This way, you have access to all the packages in your workspace, and it will make things easier with recognized dependencies and auto-completion:

$ cd ~/ros2\_ws/src/

$ code .

### Writing a minimal ROS 2 Python node

my\_first\_node.py file:

#!/usr/bin/env python3

import rclpy

from rclpy.node import Node

class MyCustomNode(Node):

def init(self):

super().init('my\_node\_name')

def main(args=None):

rclpy.init(args=args)

node = MyCustomNode()

rclpy.spin(node)

rclpy.shutdown()

if \_\_name\_\_ == '\_\_main\_\_':  
 main()

We first import ***rclpy***, the Python library for ROS 2. Inside this library, we can get the ***Node*** class.

We then create a new class that inherits from the rclpy *Node* class:

After the class, we create a main() function in which we perform the following actions:

1. Initialize ROS 2 communications with ***rclpy.init()***. This should be the first line in your main()function.
2. Create an object from the *MyCustomNode* class we wrote before. This will initialize the node.
3. Make the node spin.
4. After the node is killed, shut down ROS 2 communications with ***rclpy.shutdown()***

Finally, we also have added these two lines:

if \_\_name\_\_ == '\_\_main\_\_':

main()

This is a pure Python thing and has nothing to do with ROS 2. It just means that if you run the Python script directly, the *main()* function will be called

Before you build and run it, add one more line in the Node’s constructor, so it can do something:

self.get\_logger().info("Hello World")

This line will print “Hello World” when the node starts.

### Building the node

To build (install) the node, we need to do one more thing in the package. Open the setup.py file from the my\_py\_pkg package. Locate **entry\_points** and '**console\_scripts**' at the end of the file. For each node we want to build, we have to add one line inside the 'console\_scripts' array:

entry\_points={

'console\_scripts': [

"test\_node = my\_py\_pkg.my\_first\_node:main"

],

},

Here is the syntax:

<executable\_name> = <package\_name>.<file\_name>:<function\_name>

**Note:**

When learning ROS 2, there is a common confusion between the node name, filename, and executable name:

* **Node name**: defined inside the code, in the constructor. This is what you’ll see with the ros2 node list, or in rqt\_graph.
* **Filename**: the file where you write the code.
* **Executable name**: defined in setup.py and used with ros2 run.

The executable should now be created and installed in the workspace (it will be placed inside the install directory).

### Running the node

Run your node using ros2 run:

$ ros2 run my\_py\_pkg test\_node

### Improving the node – timer and callback

We now want to make the node print a string every second, as long as it’s alive. We will add a timer to our node. A timer will trigger a callback function at a specified rate.

class MyCustomNode(Node):

def init(self):

super().init('my\_node\_name')

self.counter\_ = 0

self.timer\_ = self.create\_timer(1.0, self.print\_hello)

def print\_hello(self):  
 self.get\_logger().info("Hello " + str(self.counter\_))  
 self.counter\_ += 1

It works because the node is spinning, thanks to rclpy.spin(node). This means that the node is kept alive, and all registered callbacks can be called during this time. What we do with create\_timer() is simply to register a callback, which can then be called when the node is spinning.

## 4.5 Creating a C++ node

Writing a C++ node

Building and running the node

## 4.6 Node template for Python and C++ nodes

### Template for a Python node

Use this code to start any new Python node:

#!/usr/bin/env python3

import rclpy

from rclpy.node import Node

class MyCustomNode(Node): # MODIFY NAME

def \_\_init\_\_(self):

super().\_\_init\_\_("node\_name") # MODIFY NAME

def main(args=None):

rclpy.init(args=args)

node = MyCustomNode() # MODIFY NAME

rclpy.spin(node)

rclpy.shutdown()

if \_\_name\_\_ == "\_\_main\_\_":

main()

Template for a C++ node

## 4.7 Introspecting your nodes

### ros2 node command line

To list all running nodes, use ros2 node list:

$ ros2 node list

/my\_node\_name

We find the name of the node, which we defined in the code.

Once we have the node name, we can get more info about it with ros2 node info :

$ ros2 node info /my\_node\_name

### Changing the node name at run time

To add any additional argument to ros2 run, first add --ros-args (only once).

Then, to rename the node, add -r \_\_node:=<new\_name>. ***-r*** means remap; you could also use --remap. For example, if we want to name the node abc, we could use this:

$ ros2 run my\_py\_pkg test\_node --ros-args -r \_\_node:=abc

# Chapter 5 - Topics – Sending and Receiving Messages between Nodes

## 5.1 What is a ROS 2 topic?

### A publisher and a subscriber

### Multiple publishers and subscribers

### Multiple publishers and subscribers inside one node

### Wrapping things up

## 5.2 Writing a topic publisher

Writing a Python publisher

To write a publisher, we need to create a node;

#### Adding a publisher to the node

For the interface, you have two choices: use an existing interface or create a custom one. To get started, we will use an existing interface.

import rclpy

from rclpy.node import Node

from example\_interfaces.msg import Int64

class NumberPublisherNode(Node):

def \_init\_\_(self):

super().\_\_init\_\_("number\_publisher")

self.number\_publisher\_ = self.create\_publisher(Int64, "number", 10)

In this method, you have to provide three arguments:

* **Topic interface**: We’ll use Int64 from the example\_interfaces package.
* **Topic name**: As defined previously, this is number.
* **Queue size**: If the messages are published too fast and subscribers can’t keep up, messages will be buffered (up to 10 here) so that they’re not lost.

#### Publishing with a timer

A common behavior in robotics is to do X action every Y seconds—for example, publish an image from a camera every 0.5 seconds, or in this case, publish a number on a topic every 1.0 second.

def \_\_init\_\_(self):

super().\_\_init\_\_("number\_publisher")

self.number\_ = 2

self.number\_publisher\_ = self.create\_publisher(Int64, "number", 10)

self.number\_timer\_ = self.create\_timer(1.0, self.publish\_number)

self.get\_logger().info("Number publisher has been started.")

def publish\_number(self):

msg = Int64()

msg.data = self.number\_

self.number\_publisher\_.publish(msg)

#### Building the publisher

To try your code, you need to install the node.

Before we do this, since we’re using a new dependency (example\_interfaces package), we also need to add one line to the package.xml file of the my\_py\_pkg package:

<depend>rclpy</depend>

<depend>example\_interfaces</depend>

To install the node, open the setup.py file from the my\_py\_pkg package and add a new line to create another executable:

entry\_points={

'console\_scripts': [

"test\_node = my\_py\_pkg.my\_first\_node:main",

"number\_publisher = my\_py\_pkg.number\_publisher:main"

],

}

#### Running the publisher

After the package has been built successfully, source your workspace and start the node:

$ source install/setup.bash # or source ~/.bashrc

$ ros2 run my\_py\_pkg number\_publisher

With the ros2 topic echo <topic\_name> command, you can subscribe to the topic directly from the subscriber and see what’s being published.

$ ros2 topic echo /number

data: 2

---

data: 2

---

As you can see, we get one new message per second, which contains a data field with a value of 2. This is exactly what we wanted to do in the code.

**Gemini explanation:**

The ros2 topic echo /number command is subscribing to the /number topic and printing (echoing) the messages it receives to the console. Let's break it down:

* **ros2 topic**: This is the base command for interacting with ROS 2 topics. It provides subcommands for various operations like publishing, subscribing, listing, and inspecting topics.
* **echo**: This subcommand tells ros2 topic to subscribe to the specified topic and "echo" or print the received messages to the standard output (your terminal). Think of it like a simple listener that displays the data being published on the topic.
* **/number**: This is the name of the topic you're subscribing to. In your example, it seems a publisher is sending messages of type (likely std\_msgs/Int32 or similar) to this topic.

### Writing a C++ publisher

## 5.3 Writing a topic subscriber

### Writing a Python subscriber

#### Creating a Python node with a subscriber

Create a new node named number\_counter inside the my\_py\_pkg package:

#!/usr/bin/env python3

import rclpy

from rclpy.node import Node

from example\_interfaces.msg import Int64

class NumberCounterNode(Node):

def \_\_init\_\_(self):

super().\_\_init\_\_("number\_counter")

self.counter\_ = 0

self.number\_subscriber\_ = self.create\_subscription(Int64, "number", self.callback\_number, 10)

self.get\_logger().info("Number Counter has been started.")

def callback\_number(self, msg: Int64):

self.counter\_ += msg.data

self.get\_logger().info("Counter: " + str(self.counter\_))

#### Running the Python subscriber

### Writing a C++ subscriber

### Running the Python and C++ nodes together

## 5.4 Additional tools to handle topics

### Introspecting topics with rqt\_graph

### The ros2 topic command line

to see the details for the interface, you can run the following command:

$ ros2 interface show example\_interfaces/msg/Int64

# some comments

int64 data

### Changing a topic name at runtime

let’s rename our topic from number to my\_number:

$ ros2 run my\_py\_pkg number\_publisher --ros-args -r number:=my\_number

### Replaying topic data with bags

**Bags** allow you to record a topic and replay it later. Thus, you can run the experiment once with the required conditions, and then replay the data just like it was recorded.

record the bag with ros2 bag record <list of topics> -o <bag\_name>. To make things more organized, I suggest that you create a bags folder and record from within this folder:

To make things more organized, I suggest that you create a bags folder and record from within this folder:

$ mkdir ~/bags

$ cd ~/bags/

$ ros2 bag record /number -o bag1

Now, you can replay the bag, which means you’ll publish on the topic exactly like it was done when recording. Stop the number\_publisher node and replay the bag with ros2 bag play <path\_to\_bag>:

$ ros2 bag play ~/bags/bag1/

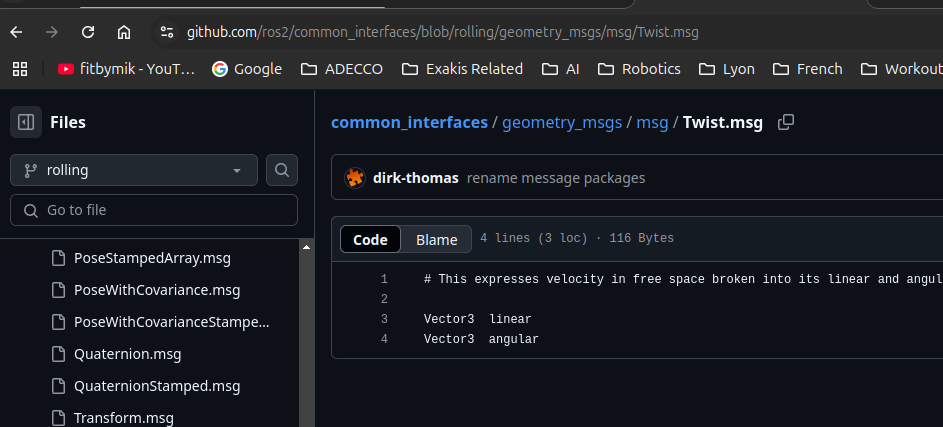
## 5.5 Creating a custom interface for a topic

Using existing interfaces

Before you start a new publisher or subscriber for a topic, take some time to think about what kind of data you want to send or receive. Then, check if an already existing interface contains what you need.

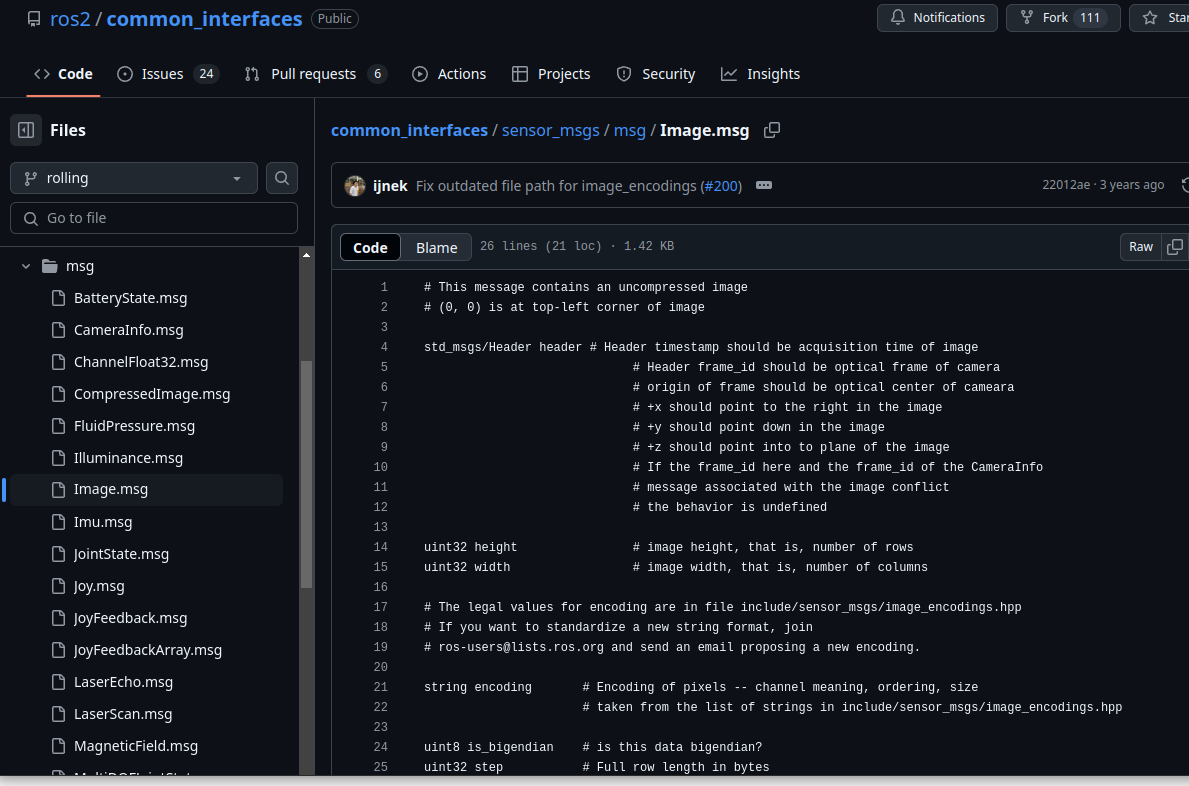
#### Where to find interfaces

Just like nodes, interfaces are organized in packages. You can find the most common packages for ROS 2 interfaces here: <https://github.com/ros2/common_interfaces>. Not all existing interfaces are listed here, but it’s already quite a lot. For other interfaces, a simple search on the internet should bring you to the corresponding GitHub repository.



In this common interfaces repository, you can find the ***Twist*** message we used with Turtlesim, inside the ***geometry\_msgs*** package. As you can see, for topic interfaces, we then have an additional msg folder, which contains all the message definitions for that package.

Now, let’s say you want to create a driver node for a camera and publish the images to a topic. If you look inside the *sensor\_msgs* package, and then inside the msg folder, you’ll find a file named *Image.msg*.



#### Using an existing interface in your code

To use this message, make sure you’ve installed the package that contains the message—in this case, sensor\_msgs. As a quick reminder, to install a ROS 2 package, you can run sudo apt install ros-<distro>-<package-name>:

$ sudo apt install ros-jazzy-sensor-msgs

Maybe the package was already installed. If not, source your environment again afterward. Then, you can find the details regarding the interface with ros2 interface show <interface>:

$ ros2 interface show sensor\_msgs/msg/Image

### Creating a new topic interface

#### Creating and setting up an interfaces package

Before we create any topic interface (message), we need to create a new package and set it up for building interfaces.

A common practice when naming this interface package is to start with the name of your application or robot and add the *\_interfaces* suffix. So, if your robot is named *abc*, you should use *abc\_interfaces*.

$ cd ~/ros2\_ws/src/

$ ros2 pkg create my\_robot\_interfaces

We need to set this new package up and modify a few things so it can build messages. Go into the package, remove the src and include directories, and create a new msg folder:

$ cd my\_robot\_interfaces/

$ rm -r src/ include/

$ mkdir msg

Now, open the package.xml file for this package. add the following three lines:

<build\_depend>rosidl\_default\_generators</build\_depend>

<exec\_depend>rosidl\_default\_runtime</exec\_depend>

<member\_of\_group>rosidl\_interface\_packages</member\_of\_group>

Then, open the CMakeLists.txt file. After find\_package(ament\_cmake REQUIRED), and before ament\_package(), add the following lines (you can also remove the if(BUILD\_TESTING) block):

find\_package(rosidl\_default\_generators REQUIRED)

rosidl\_generate\_interfaces(${PROJECT\_NAME}

# we will add the name of our custom interfaces here

)

ament\_export\_dependencies(rosidl\_default\_runtime)

There’s not much to understand about these lines you’re adding. They will find some dependencies (rosidl packages) and prepare your package so that it can build interfaces. At this point, your package is ready and you can add new interfaces.

#### Explanation from Claude

Let me explain these CMake commands commonly used in ROS 2 for generating message interfaces:

**find\_package**(rosidl\_default\_generators REQUIRED)

* 1. This line finds and loads the *rosidl\_default\_generators* package, which provides tools for generating ROS 2 message interfaces
  2. The REQUIRED flag means CMake will throw an error if it can't find this package
  3. This package is essential for creating custom message, service, and action interfaces in ROS 2

rosidl\_generate\_interfaces(${PROJECT\_NAME} ...)

* 1. This command generates the code for your custom interfaces
  2. ${PROJECT\_NAME} is a CMake variable that contains your package name. The commented section *“# we will add the name of our custom interfaces here”* is where you would list your interface files, for example:

rosidl\_generate\_interfaces(${PROJECT\_NAME}  
 "msg/CustomMessage.msg"  
 "srv/CustomService.srv"  
 "action/CustomAction.action"  
)

When you add interface files to this command, ROS 2 will automatically:

* Generate the necessary C++, Python, and other language bindings
* Create header files and source code for your interfaces
* Set up the build system to compile these generated files

#### Creating and building a new topic interface

To make things simple here, we will start with only built-in types. Write the following inside the message file:

int64 version

float64 temperature

bool are\_motors\_ready

string debug\_message

For each field, we provide the data type, and then the name of the field.

To build the message, you simply have to add one line to CMakelists.txt, specifying the relative path to the message file:

rosidl\_generate\_interfaces(${PROJECT\_NAME}

"msg/HardwareStatus.msg"

)

#### Using your custom message in your code

First, open the package.xml file from the my\_py\_pkg package and add a dependency to my\_robot\_interfaces:

<depend>rclpy</depend>

<depend>example\_interfaces</depend>

<depend>my\_robot\_interfaces</depend>

Then, for Python, do the following:

Import the message by by adding the following import line in your code:

from my\_robot\_interfaces.msg import HardwareStatus

Create a message in your code, like so:

msg = HardwareStatus()

msg.temperature = 34.5

## 5.6 Topic challenge – closed-loop control

Challenge

Solution

## 5.7 Summary

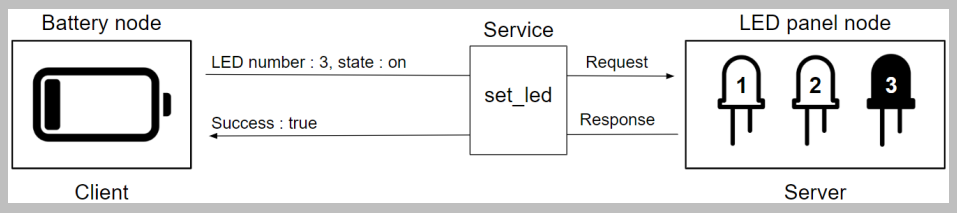
# Chapter 6 - Services – Client/Server Interaction between Nodes

## 6.1 What is a ROS 2 service?

### A server and a client

### Multiple clients for one service

### Another service example with robotics



### Wrapping things up

On top of topics, ROS 2 nodes can use services to communicate with each other.

When should you use topics versus services? You should use topics to publish unidirectional data streams and services when you want to have a client/server type of communication.

For example, if you want to continuously send a velocity command to a robot 10 times per second, or send the data you read from a sensor, you will use topics. If you want to have a node perform quick computations or do some actions on demand (enabling/disabling a motor, starting/stopping a robot), then you would use services.

## 6.2 Creating a custom service interface

Finding an existing interface for our service

### Creating a new service interface

To create a service interface, just like for a topic interface, you need to create and configure a package dedicated to interfaces.

First, navigate inside the my\_robot\_interfaces package (where you already have a msg folder) and create a new srv folder: $ cd ~/ros2\_ws/src/my\_robot\_interfaces/ $ mkdir srv

Use .srv for the file extension.

Open the new srv file and write the definition for the service interface. One very important thing to do here is add three dashes (---) and put the request definition on top, and then the response definition below the dashes.

int64 reset\_value

---

bool success

string message

Go back to the CMakeLists.txt of my\_robot\_interfaces package

rosidl\_generate\_interfaces(${PROJECT\_NAME}

"msg/HardwareStatus.msg"

"srv/ResetCounter.srv"

)

## 6.3 Writing a service server

### Writing a Python service server

#### Importing a service interface

Now, import the dependency into your code (number\_counter.py):

#!/usr/bin/env python3

import rclpy

from rclpy.node import Node

from example\_interfaces.msg import Int64

from my\_robot\_interfaces.srv import ResetCounter

Adding a service server to the node

def init(self):

...

self.reset\_counter\_service\_ = **self.create\_service**(ResetCounter, "reset\_counter", self.callback\_reset\_counter)

In this method, you must provide three arguments:

* **Service interface**: This is the *ResetCounter* class we have imported.
* **Service name**: Whenever you create a service server, you’re creating the service itself, so you decide what its name will be. As best practice, start with a verb. Since we want to reset the counter, we’ll simply name it reset\_counter.
* **Service callback**:

#### Validating the request

### Writing a C++ service server

## 6.4 Writing a service client

### Writing a Python service client

Open the file and start by importing the interface:

from my\_robot\_interfaces.srv import ResetCounter

In the node’s constructor, create a service client:

def \_\_init\_\_(self):

super().\_\_init\_\_("reset\_counter\_client")

self.client\_ = self.create\_client(ResetCounter, "reset\_counter")

To create a service client, we use the ***create\_client()*** method from the Node class.

Then, to call the service, we create a new method:

def call\_reset\_counter(self, value):

while not self.client\_.wait\_for\_service(1.0):

self.get\_logger().warn("Waiting for service...")

request = ResetCounter.Request()

request.reset\_value = value

future = self.client\_.call\_async(request)

future.add\_done\_callback(self.callback\_reset\_counter\_response)

Here are the steps to make a service call:

1. Make sure the service is up and running with wait\_for\_service(). This function will return True as soon as the service has been found, or return False after the provided timeout, which is 1.0 seconds here.
2. Create a request object from the service interface.
3. Fill in the request fields.
4. Send the request with call\_async(). This will give you a Python Future object.
5. Register a callback for when the node receives the response from the server. To process the response from the service, add a callback method:

def callback\_reset\_counter\_response(self, future):

response = future.result()

self.get\_logger().info("Success flag: " + str(response.success))

self.get\_logger().info("Message: " + str(response.message))

In the callback, we get the response with *future.result()* .

### Running the client and server nodes together

### Writing a C++ service client

## 6.5 Additional tools to handle services

To see all commands for ROS 2 services, type ros2 service -h.

### Listing and introspecting services

to list all services, run the following command:

$ ros2 service list

Once you have the service name you want, you can get the service interface with ros2 service type :

$ ros2 service type /reset\_counter

my\_robot\_interfaces/srv/ResetCounter

From this, you can see the details inside the interface:

$ ros2 interface show my\_robot\_interfaces/srv/ResetCounter

int64 reset\_value

---

bool success

string message

### Sending a service request

First, you must know the service name and interface. Then, use the ros2 service call <service\_name> <interface\_name> "<request\_in\_json>" command.

$ ros2 service call /reset\_counter my\_robot\_interfaces/srv/ResetCounter "{reset\_value: 7}"

waiting for service to become available...

requester: making request: my\_robot\_interfaces.srv.ResetCounter\_ Request(reset\_value=7)

response:

my\_robot\_interfaces.srv.ResetCounter\_Response(success=True, message='Success')

### Changing a service name at runtime

to rename a service, add -r followed by :=.

Eg.

$ ros2 run my\_py\_pkg number\_counter --ros-args -r reset\_counter:=reset\_counter1

## 6.6 Service challenge – client and server

Challenge

Solution

## 6.7 Summary

# Chapter 7 - Actions – When Services Are Not Enough

Imagine that you have a mobile robot with two wheels.

This node would also be able to receive commands, such as Move to (x, y) coordinates.

With what we know so far, ROS 2 services seem to be a good option. In this server node, you could implement a /move\_robot service that will receive coordinates from a client.

Well, moving a physical part of a robot in space can take some time. It could be a fraction of a second in some cases, but also maybe a few seconds, or even a few minutes. The point is that the service execution could take a significant amount of time.

With that said, while the robot is moving, there are a few things you may want to do, and those things are missing when using services:

* Since the execution is taking some time, it would be nice to get some feedback from the server. With a service, the client has no idea of what’s happening on the server side. So, the client is completely blind and needs to wait for the response to get some information.
* How can you cancel the current execution? That would seem a reasonable feature to have. After you start the execution on the server side, the client may want to cancel it. For example, let’s say the client node is also monitoring the environment with a camera. If an obstacle is detected, the client could ask the server to stop the execution. With what we have for now, the client can’t do anything but wait for the server to finish the execution.
* Here’s the last point for now, although we could find more: how could the server correctly handle multiple requests? Let’s say you have two or more clients, each one sending a different request. How can you possibly choose between those requests on the server? How can the server refuse to execute a request, or choose to replace a request with a new one, without finishing the first request?

## 7.1 What is a ROS 2 action?

### Why actions?

### How do actions work?

### Wrapping things up

## 7.2 Creating a custom action interface

### Defining the application and the interface we need

### Creating a new action interface

## 7.3 Writing an action server

Writing a Python action server

Writing a C++ action server

## 7.4 Writing an action client

Writing a Python action client

Creating an action client

Writing a C++ action client

## 7.5 Taking advantage of all the action mechanisms

### Adding the feedback mechanism

### Adding the cancel mechanism

## 7.6 Additional tools to handle actions

### Listing and introspecting actions

### Sending a goal from the Terminal

### Topics and services inside actions

## 7.7 Summary

# Chapter 8 - Parameters – Making Nodes More Dynamic

## 8.1 What is a ROS 2 parameter?

### Why parameters?

### Example of a node with parameters

### ROS 2 parameters – wrapping things up

## 8.2 Using parameters in your nodes

### Declaring, getting, and using parameters with Python

Before using a parameter, we need to declare it. Where should we declare parameters? We will do that in the node’s constructor, before everything else. To declare a parameter, use the ***declare\_parameter()*** method from the Node class. You will provide two arguments:

* **Parameter name**: This is the name that you will use to set the parameter’s value at runtime
* **Default value**: If the parameter’s value is not provided at runtime, this value will be used

Open the number\_publisher.py file, and let’s declare two parameters in the constructor:

self.declare\_parameter("number", 2)

self.declare\_parameter("publish\_period", 1.0)

Now, declaring a parameter means that it exists within the node, and you can set a value from the outside. However, in your code, to be able to use the parameter, it’s not enough to declare it. After doing that, you need to get the value.

For this, you will use the ***get\_parameter()*** method, and provide the parameter’s name as an argument. Then, you can access the value with the value attribute:

self.number\_ = self.get\_parameter("number").value

self.timer\_period\_ = self.get\_parameter("publish\_period").value

### Providing parameters at runtime

To provide a parameter’s value with the ros2 run command, follow the next steps:

1. You will first start your node with ros2 run .
2. Then, to add any argument after this command, you have to write --ros-args (only once).
3. To specify a parameter’s value, write -p :=. You can add as many parameters as you want.

Let’s say we want to start the node and publish the number 3 every 0.5 seconds. In that case, we’d run the following command:

$ ros2 run my\_py\_pkg number\_publisher --ros-args -p number:=3 -p publish\_period:=0.5

### Parameters with C++

## 8.3 Storing parameters in YAML files

As your ROS 2 application grows, so will the number of parameters. Adding 10 or more parameters from the command line is not really an option anymore.

Fortunately, you can use YAML files to store your parameters, and you can load these files at runtime.

### Loading parameters from a YAML file

let’s just create a new file in our home directory

$ touch number\_params.yaml

Edit this file and add parameters for the /number\_publisher node:

/number\_publisher:

ros\_\_parameters:

number: 7

publish\_period: 0.8

Once you’ve written this file, you can load the parameters with the ***--params-file*** argument:

$ ros2 run my\_py\_pkg number\_publisher --ros-args --params-file ~/number\_params.yaml

### Parameters for multiple nodes

### Recapping all parameters’ data types

## 8.4 Additional tools for handling parameters

### Getting parameters’ values from the terminal

### Exporting parameters into YAML

### Setting a parameter’s value from the terminal

### Parameter services

## 8.5 Updating parameters with parameter callbacks

In this section, you will learn how to implement a parameter callback for Python. This callback will be triggered whenever a parameter’s value has been changed, and we will be able to get the new value in the code.

Parameter callbacks are a great way to change a setting in your node without having to create yet another service.

Python parameter callback

Let’s write our first Python parameter callback.

Open the number\_publisher.py file and register a parameter callback in the node’s constructor:

self.add\_post\_set\_parameters\_callback(self.parameters\_callback)

We also add a new import line:

from rclpy.parameter import Parameter

Then, we implement the callback method:

def parameters\_callback(self, params: list[Parameter]):

for param in params:

if param.name == "number":

self.number\_ = param.value

In this callback, you receive a list of Parameter objects. For each parameter, you can access its name, value, and type. With a for loop, we go through each parameter we get and set the corresponding values in the code.

### C++ parameter callback

## 8.6 Parameter challenge

Challenge

Solution

## 8.7 Summary

# Chapter 9 - Launch Files – Starting All Your Nodes at Once

## 9.1 What is a ROS 2 launch file?

Why launch files?

As your ROS 2 application starts to grow, so does the number of nodes and parameters. For example, a ROS stack I developed for a robotic arm had more than 15 nodes and 200 parameters. Imagine opening 15 terminals and starting all the nodes one by one with all the correct values for parameters. This would quickly become a nightmare.

### Example of a launch file with seven nodes

## 9.2 Creating and installing an XML launch file

You will now create your first launch file. We will start with XML. Later in this chapter, we will also write Python launch files and compare the two languages

### Setting up a package for launch files

### Writing an XML launch file

<launch>

<node pkg="my\_py\_pkg" exec="number\_publisher"/>

<node pkg="my\_cpp\_pkg" exec="number\_counter"/>

</launch>

### Installing and starting a launch file

You now have to install your new launch file before you can start using it. As we are starting nodes from the my\_py\_pkg and my\_cpp\_pkg packages, we need to add the dependencies in the package.xml file of the my\_robot\_bringup package:

<exec\_depend>my\_py\_pkg</exec\_depend>

<exec\_depend>my\_cpp\_pkg</exec\_depend>

Now, we can install the launch file. To do so, you just need to build your package:

$ cd ~/ros2\_ws/

$ colcon build --packages-select my\_robot\_bringup

Then, source your environment, and use the ros2 launch command-line tool to start the launch file. The full command is ros2 launch <package\_name> <launch\_file\_name>:

$ ros2 launch my\_robot\_bringup number\_app.launch.xml

## 9.3 Creating a Python launch file – XML or Python for launch files?

### Writing a Python launch file

For Python launch files, you will use the .launch.py extension. Create a new file named number\_app.launch.py. Here is the code required to start the number\_publisher and number\_counter nodes:

from launch import LaunchDescription

from launch\_ros.actions import Node

def generate\_launch\_description():

ld = LaunchDescription()

number\_publisher = Node( package="my\_py\_pkg", executable="number\_publisher" )

number\_counter = Node( package="my\_cpp\_pkg", executable="number\_counter" )

ld.add\_action(number\_publisher)

ld.add\_action(number\_counter)

return ld

The first thing you will notice is that the code is much, much longer than the XML one.

required steps to write a Python launch file:

1. The launch file must include a generate\_launch\_description() function. Make sure you don’t make any typos.
2. In this function, you will need to create and return a LaunchDescription object. You can get this from the launch module.
3. To add a node in the launch file, you create a Node object (from launch\_ros.actions) and specify the package and executable name. Then, you can add this object to the LaunchDescription object.

### XML versus Python for launch files

#### Including a launch file inside another launch file

To include another launch file, use an <include> tag:

<launch>

<include file="$(find-pkg-share my\_robot\_bringup)/launch/number\_app.launch.py" />

</launch>

## 9.4 Configuring nodes inside a launch file

### Renaming nodes and communications

### Parameters in a launch file

Here is an example, where we set the number and publish\_period parameters for the number\_ publisher node:

<node pkg="my\_py\_pkg" exec="number\_publisher">

<param name="number" value="3" />

<param name="publish\_period" value="1.5" />

</node>

It will work the same as adding -p := after the ros2 run command.

### Namespaces

## 9.5 Launch file challenge

Challenge

Solution

## 9.6 Summary

# Part 3: Creating and Simulating a Custom Robot with ROS 2

# Chapter 10 - Discovering TFs with RViz

In ROS, a TF is the transformation between two frames in 3D space. TFs will be used to track the different coordinate frames of a ROS robot (or system with multiple robots) over time. They are used everywhere and will be the backbone of any robot you create.

**RViz** is a 3D visualization package.

## 10.1 Technical requirements

## 10.2 Visualizing a robot model in RViz

### Installation and setup

there is no need to install RViz. It was already included when you installed ROS 2 at the beginning of the book (with the sudo apt install ros-<distro>-desktop command).

To visualize TFs for a robot model on RViz, we will install a new ROS package named ***urdf\_tutorial***. This package contains some existing launch files and robot model files (how to create a robot model will be the focus of the next chapter).

$ sudo apt install ros-<distro>-urdf-tutorial

Then, so that you can use the package, make sure you source the environment or simply open a new terminal.

### Starting RViz with a robot model

The urdf\_tutorial package contains a launch file, named display.launch.py, that will start RViz and load a robot model into it. For now, we will just use it, and in the following chapters, we will understand how this process works so we can replicate it.

So, we need to start this launch file and also load a robot model. Where will we get one? There are some existing models in the urdf\_tutorial package. To find them, navigate to the share directory where the package was installed, and you will find a urdf folder under the package name:

$ cd /opt/ros/<distro>/share/urdf\_tutorial/urdf/

A **Unified Robot Description Format (URDF)** file is basically the description of a robot model.

In the urdf folder, you can find several robot model files:

$ ls

01-myfirst.urdf 04-materials.urdf 07-physics.urdf

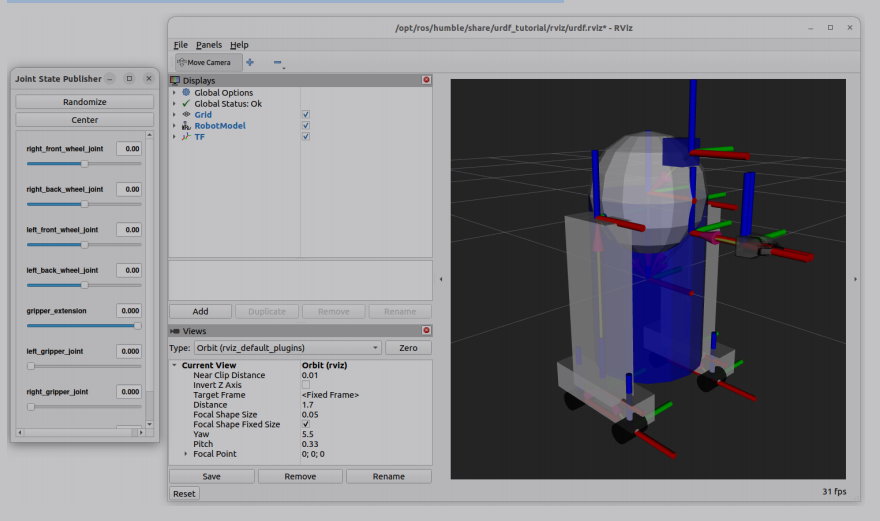
02-multipleshapes.urdf 05-visual.urdf 08-macroed.urdf.xacro

03-origins.urdf 06-flexible.urdf

You can now start a robot model in RViz by launching the display.launch.py file and add the path to the robot model with an additional ***model*** argument after the launch file:

$ ros2 launch urdf\_tutorial display.launch.py model:=/opt/ros/jazzy/share/urdf\_tutorial/urdf/07-physics.urdf

After running the command, you should see something like this:



You will get two windows: the main one (RViz) with the robot model, and a Joint State Publisher window with some cursors.

## 10.3 What are TFs?

There are two main parts in a robot model: links and TFs.

### Links

Have a look at the menu on the left side of the **RViz** window. There, you will see, in blue bold letters, **RobotModel** and **TF**. As you can see, you can enable or disable both menus.

Disable **TF**, keep **RobotModel**, and expand the menu. There, you can find a submenu called **Links**.

Check and uncheck some boxes. As you can see from this menu, a link is one rigid part (meaning one solid part with no articulation) of the robot. Basically, in ROS, a robot model will consist of a collection of rigid parts put together.

In this example, links are represented by basic shapes: boxes, cylinders, and spheres.

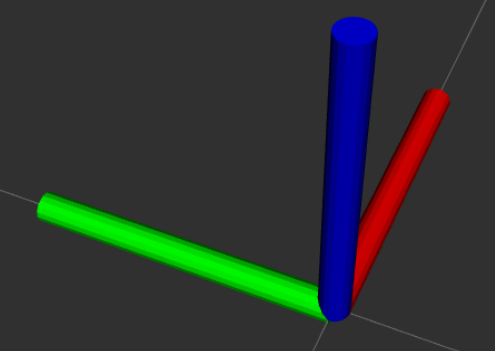
### TFs

Let’s now check the **TF** box. You can keep **RobotModel** checked or unchecked. Inside the **TF** menu, you have a submenu called **Frames**, and you can also enable or disable each frame for the robot.

The axes you see here (red, green, and blue coordinate systems) represent the frames, or basically the origin of each link of the robot.

Coordinate systems follow the right-hand rule in ROS. Following Figure 10.4, you have the following:

* X axis (red) pointing forward
* Y axis (green) pointing 90 degrees to the left
* Z axis (blue) pointing up



## 10.4 Relationship between TFs

### Parent and child

Each TF will be connected to another TF, with a parent/child relationship. To see one, you can, for example, disable all TFs on RViz, and only check the base\_link and gripper\_pole frames.

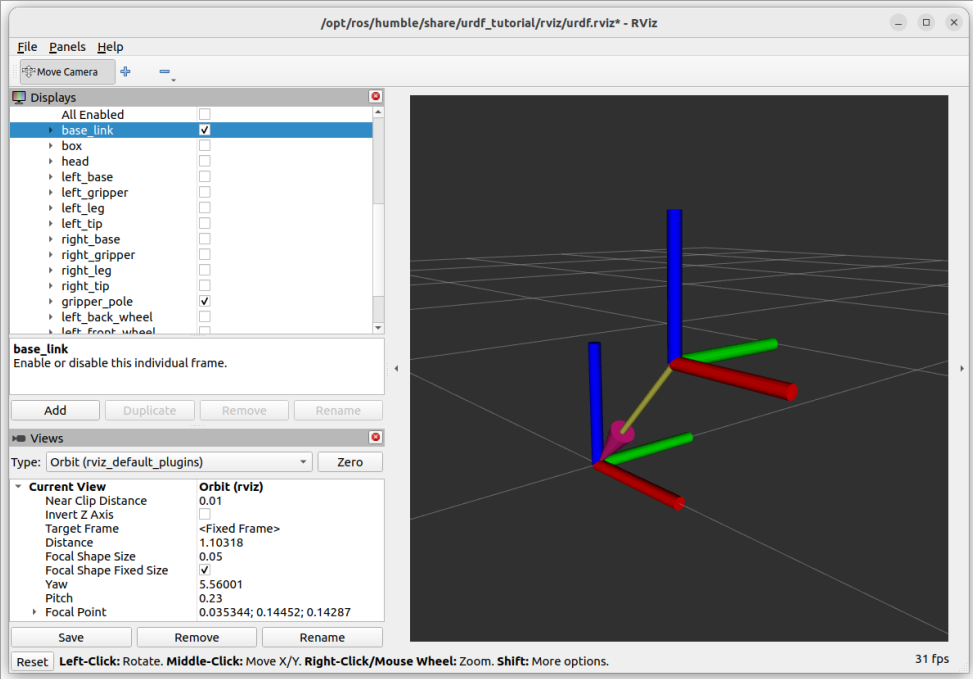


Figure 10.5 – The relationship between two frames

As you can see in this example, an arrow is going from the *gripper\_pole* frame to the *base\_link* frame. This means that *gripper\_pole* is the child of *base\_link* (or, *base\_link* is the parent of *gripper\_pole*).

The order of those relationships is very important. If you move *gripper\_pole* relative to *base\_link* (the *gripper\_extension* cursor in the Joint State Publisher window), then anything that’s attached to *gripper\_pole* (meaning children of *gripper\_pole*) will also move with it.

That makes sense: when you rotate your elbow, your forearm is moving, but also your wrist, hand, and fingers. They don’t move relative to the forearm, but as they are attached to it, they move relative to the arm.

### The /tf topic

At this point, you might think that what we did in Part 2 of this book has nothing to do with what we are doing now. Well, everything we have seen here is still based on nodes, topics, and so on.

Let’s list all nodes:

$ ros2 node list

/joint\_state\_publisher

/robot\_state\_publisher

/rviz

/transform\_listener\_impl\_5a530d0a8740

You can see that **RViz** is actually started as a node (*rviz*).

Now, let’s list all topics:

$ ros2 topic list

/joint\_states

/parameter\_events

/robot\_description

/rosout

/tf

/tf\_static

You can subscribe to the topic from the terminal with the following:

$ ros2 topic echo /tf

If you do so, you will receive lots of messages. Here is an extract:

transforms:

- header:

stamp:

sec: 1719581158

nanosec: 318170246

frame\_id: base\_link

child\_frame\_id: gripper\_pole

transform:

translation:

x: 0.19

y: 0.0

z: 0.2

rotation:

x: 0.0

y: 0.0

z: 0.0

w: 1.0

This extract matches what we previously saw on RViz. It represents the transformation between base\_link and gripper\_pole.

### Visualizing the TF tree

To do that, you will need to use the tf2\_tools package. Make sure it is installed:

$ sudo apt install ros-jazzy-tf2-tools

Don’t forget to source the environment after installing the package. Now, keep the robot running on RViz, and execute this command in a second terminal:

$ ros2 run tf2\_tools view\_frames

it will listen to the /tf topic for five seconds. After this, the command exits with a big log that you can ignore.

You will get two new files, in the same directory as where you ran the command.

Open the PDF file. You will see something like this:

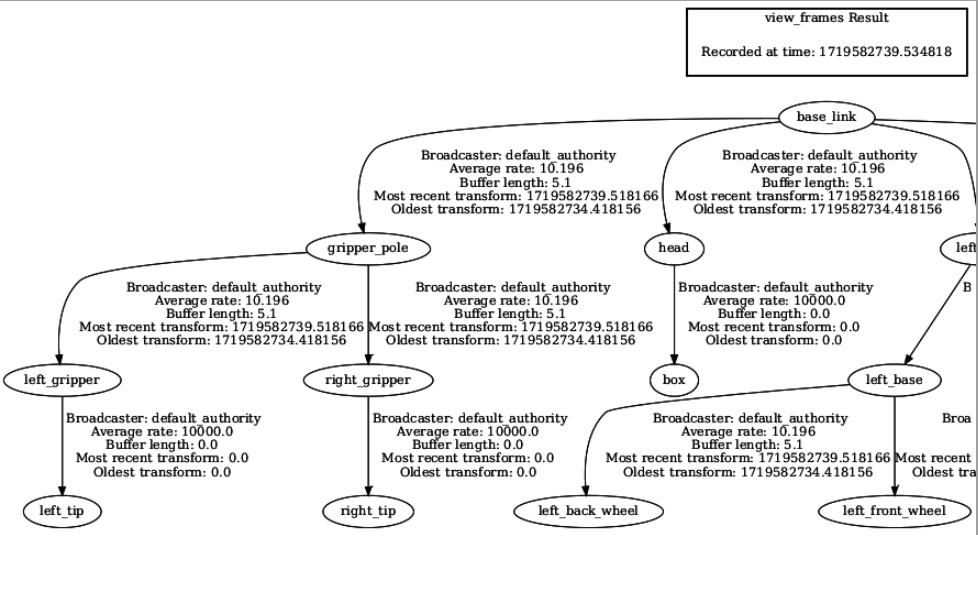


Figure 10.6 – TF tree for a robot

## 10.5 What problem are we trying to solve with TFs?

### What we want to achieve

For a robotics application to work, we want to keep track of each 3D coordinate frame over time. We need a structured tree for all the frames of the robot (or robots).

There are two components here: we need to know where things are and when the transformations happened. If you remember, when we checked the /tf topic, you could see that for each parent and child frame, we had a transformation (translation and rotation in 3D space), and we also had a timestamp.

Eg.

If you have an application with a robotic arm and a camera, then where is the camera relative to the base of the robot? And to the hand of the robot, so that the arm can correctly pick up and place objects that were detected by the camera?

So, with TFs, we want to know the following:

* How frames are placed relative to each other
* How they move relative to each other and over time

### Frames according to Grok

Imagine a robot with multiple components—like a base, a camera, or a robotic arm. Each of these components might have its own local coordinate system (or frame). For example, the camera might have a frame where the origin is at its lens, and the axes are aligned with its field of view. The robot’s base might have another frame, with its origin at the center of the robot and axes aligned with its body. "Frames" in ROS allow you to keep track of how these different coordinate systems relate to one another.

### How to compute TFs

What is a transformation exactly? A transformation is the combination of a translation and a rotation in space.

As we are working in 3D, we have three components for the translation (x, y, z), and three components for the rotation (x, y, z). To find a transformation between two frames, you will need to compute those six elements, using 3x3 matrices.

I won’t go into any mathematical details here, but you can probably guess that it won’t be an easy task. Also, you need to compute this transformation for each frame, relative to the other frame. This increases the complexity.

For example, let’s say you need to know where left\_front\_wheel is relative to base\_link. Following the previous TF tree (open the PDF again), you can see that you need to follow this order: base\_link, left\_leg, left\_base, and left\_front\_wheel.

You will need to compute three transformations in a row so that you can get this base\_link to left\_front\_wheel transformation. You will have to repeat this for each frame relative to all other frames (the complexity then increases a lot as you add more frames), and track these over time.

This sounds like a lot of work. Fortunately, we don’t have to do any of that, thanks to the ROS TF functionality. There is a library called **tf2**, and it already does that for us.

In the end, the biggest challenge with TFs is to understand how they work. You will mostly not use TFs directly in your application. Several packages will handle that for you. The only thing we need to do is to provide a robot description that specifies all the links and TFs for the robot. Then, using a package named robot\_state\_publisher, ROS will automatically publish TFs for us.

## 10.6 Summary

# Chapter 11 - Creating a URDF for a Robot 253

URDF mens ***Unified Robot Description Format***.

Basically, a URDF file will contain a description of all the elements of a robot. You will define each link (rigid part) of the robot. Then, to create relationships between the links, you will add some joints, which will be used to generate the TFs.

Ulisses’s Note: Used to generate TFs. It can be visualized using RViz.

In this chapter: We will create a mobile base with two wheels.

## 11.1 Creating a URDF with a link

### Setting up a URDF file

A URDF file is simply an XML file with the .urdf extension.

Let’s create a URDF file named *my\_robot.urdf* .

Here is the minimum code you have to write inside a URDF file:

<?xml version="1.0"?>

<robot name="my\_robot">

</robot>

### Creating a link

#### Basic code for a link

To get started, let’s imagine the main base of a robot, represented as a box. The box is 60cm x 40cm x 20cm, or 0.6m x 0.4m x 0.2m.

box: long x width x height

Here is the code for this first link:

<robot name="my\_robot">

<link name="base\_link">

<visual>

<geometry>

<box size="0.6 0.4 0.2" />

</geometry>

<origin xyz="0 0 0" rpy="0 0 0" />

</visual>

</link>

</robot>

The **<link>** tag defines the link. All the properties for this link must be inside the tag. You also have to provide a name attribute for the link. As a convention, for the first link, we use base\_link.

Then, inside this tag, we have the **<visual>** tag. If you want to define a visual appearance for the link (rigid part), you can do so with this tag. Inside, you will have the following:

* **<geometry>**: This will define the shape of the link. Here, we use the **<box>** tag and provide the dimensions with the size attribute.
* **<origin>**: This tag is quite important, as it defines the origin of the visual relative to the origin of the link. The origin contains six elements for translation and rotation.

Note: The origin of rotation is written as **rpy**. This means ***roll***, ***pitch***, ***yaw***. It’s the same as x, y, and z, but using different names. Roll, pitch, and yaw are quite frequently used for aviation.

#### Visualizing the URDF in RViz

To test/visualize our urdf file:

$ ros2 launch urdf\_tutorial display.launch.py model:=/home/<user>/my\_robot.urdf

We will then see a box (red color by default) inside RViz. You will also have the ***Joint State Publisher*** window, empty, with no cursor.

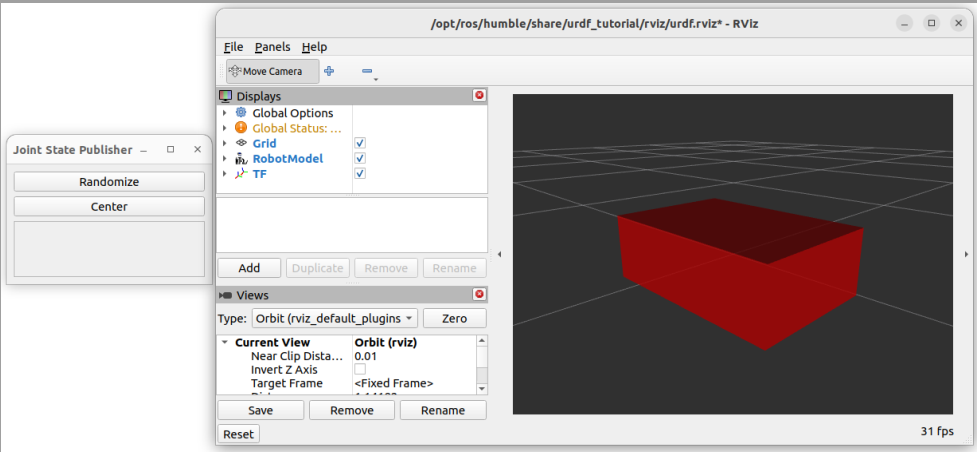


Figure 11.1 – Visualization of your URDF in Rviz

#### Modifying the origin of the visual

The link we have created is perfectly fine. However, we will offset the visual a bit so that the origin is not centered in the middle of the box, but instead, at the bottom of the box.

We have defined the height as 0.2 m, so we need to offset the visual by 0.1 m.

<origin xyz="0 0 0.1" rpy="0 0 0" />

### Customizing the link visual

#### Different shapes for a link

On top of those basic shapes, you can also use custom meshes that you export from CAD software, such as *SolidWorks*, *Blender*, and so on. You can use *STL* and *Collada* files, with **.stl** and **.dae** extensions, respectively. Setting up those files is not complicated but requires you to properly package your application around the URDF

#### Link color

To add a color to a link, you first need to create a **<material>** tag with a name. Then, you can use the **color** in your link visual.

Here is the complete code to make the link green:

<?xml version="1.0"?>

<robot name="my\_robot">

<material name="green">

<color rgba="0 0.6 0 1" />

</material>

<link name="base\_link">

<visual>

<geometry>

<box size="0.6 0.4 0.2" />

</geometry>

<origin xyz="0 0 0.1" rpy="0 0 0" />

<material name="green" />

</visual>

</link>

</robot>

## 11.2 The process of assembling links and joints

let’s add another link, and connect them with a joint. This joint will be used to generate a TF.

Properly assembling two links with a joint is the main problem anybody faces when learning URDF. There are several origins and axes you can modify, and getting two parts to be correctly placed between each other, with the correct movement, can be challenging.

### Step 1 – adding a second link

For this example, we want to add a cylinder (radius: 0.1 m, length: 0.3 m) on top of the box.

<material name="gray">

<color rgba="0.7 0.7 0.7 1" />

</material>

...

<link name="shoulder\_link">

<visual>

<geometry>

<cylinder radius="0.1" length="0.3" />

</geometry>

<origin xyz="0 0 0" rpy="0 0 0" />

<material name="gray" />

</visual>

</link>

Note: all links in a URDF need to be related to each other with a parent/child relationship

### Step 2 – adding a joint

To define how two links are connected, you need to add a joint.

<joint name="base\_shoulder\_joint" type="fixed">

<parent link="base\_link" />

<child link="shoulder\_link" />

<origin xyz="0 0 0" rpy="0 0 0" />

</joint>

**type**: For now, set it as fixed, which means that the two links won’t move between each other.

Inside the <joint> tag, you then have three more tags:

* **<parent>**: This is the parent link. You have to write the exact name of the link with the link attribute.
* **<child>**: You will write the exact name of the child link with the link attribute.
* **<origin>**: This will define the origin of the child link relative to the origin of the parent link.

Advice: start by putting zeros everywhere in the origin (for both xyz and rpy).

#### Claude explanation - What is the relation between joints and frames?

1 - A joint defines:

* HOW one frame can move relative to another (rotation, sliding, etc.)
* WHERE the motion occurs (through the joint origin)
* WHAT are the allowed movements (through joint limits)

2 - Frame Transformation Chain:

Each joint creates a transform between its parent and child frames

Think of it like this: Joints are the "rules" for how frames can move relative to each other. When you move a joint, you're really updating the mathematical relationship between two frames.

### Step 3 – fixing the joint origin

The first thing to do is to modify the <origin> tag of the joint, so you get the frame of the child link correctly placed. This is what matters the most. You will first fix the joint origin, and then, and only then, fix the visual origin.

Then, ask yourself this question: where should be the frame for the shoulder\_link, relative to the frame of the base\_link?

We want the shoulder\_link to be on top of the box, so we need to move the frame by the height of the box; here, that’s 0.2 m. There is no rotation needed for this joint, just a translation.

So, you can now modify the <origin> tag inside the <joint> tag:

<origin xyz="0 0 0.2" rpy="0 0 0" />

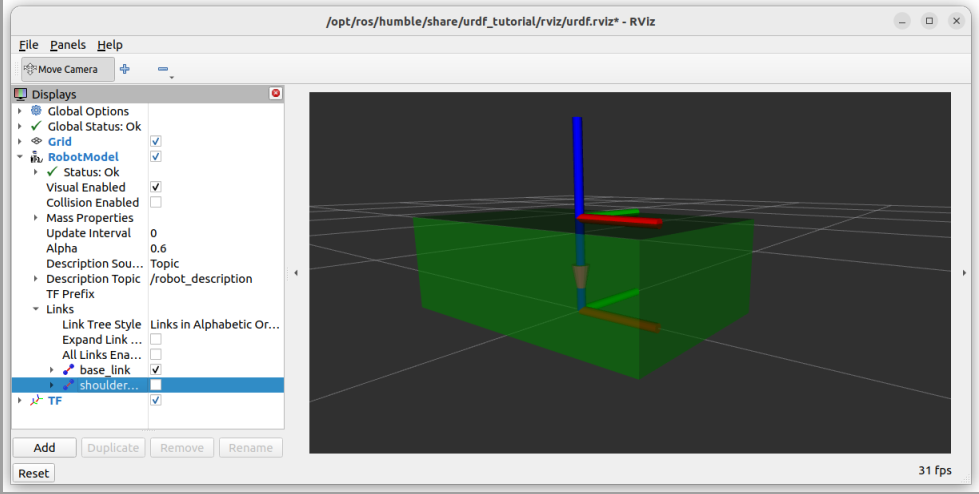


Figure 11.4 – Setting the joint origin without the visual

Great, it seems that the frame for the shoulder\_link is in the right place: on top of the box.

### Step 4 – setting up the joint type

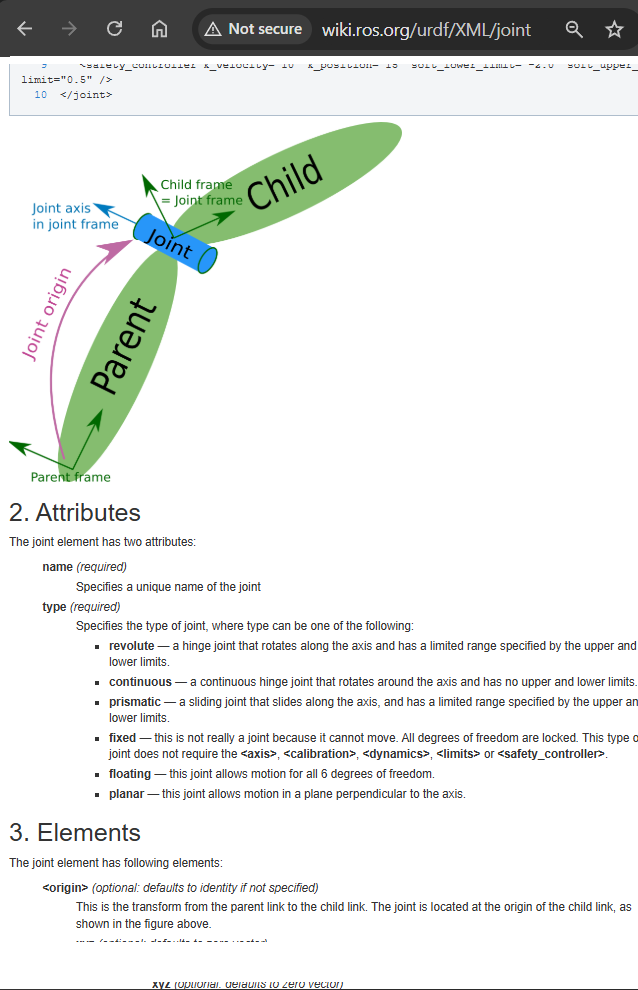
To keep things simple for the previous explanation, we have set the joint type as *fixed*, which means that the two links are not moving relative to each other.

Lots of joints you will create will be like this. For example, if you place a sensor (camera, lidar) on your robot, the sensor won’t move.

Joint types:

* **Fixed**: As previously mentioned, this is if you have two parts that are not moving
* **Revolute**: A rotation with a minimum and maximum angle, for example, in robotic arms
* **Continuous**: An infinite rotation, usually used for wheels
* **Prismatic**: If you need to make a part of your robot slide (only translation, no rotation)

You can find the complete reference for all joint types at <http://wiki.ros.org/urdf/XML/joint>.



let’s say that the shoulder link is rotating

<joint name="base\_shoulder\_joint" type="revolute">

<parent link="base\_link" />

<child link="shoulder\_link" />

<origin xyz="0 0 0.2" rpy="0 0 0" />

<axis xyz="0 0 1" />

<limit lower="-3.14" upper="3.14" velocity="100" effort="100"/>

</joint>

When choosing revolute, we first need to define which axis will be rotating. As we chose z (if you look at Figure 11.4, we want to rotate around the blue axis), we write "0 0 1", which means: no rotation on x and y, and a rotation on z.

We set the revolution between -180 and +180 degrees (about -3.14 and 3.14 radians). We also have to specify a value for the ***velocity*** and ***effort*** limits. Those two will usually be overridden by other ROS nodes. Set them to 100 by default; it won’t be important here.

*From the ros joint web page:*

**<limit>** (required only for revolute and prismatic joint)

An element can contain the following attributes:

* **lower** (optional, defaults to 0): An attribute specifying the lower joint limit (in radians for revolute joints, in metres for prismatic joints). Omit if joint is continuous.
* **upper** (optional, defaults to 0): An attribute specifying the upper joint limit (in radians for revolute joints, in metres for prismatic joints). Omit if joint is continuous.
* **effort** (required): An attribute for enforcing the maximum joint effort (|applied effort| < |effort|). See safety limits.
* **velocity** (required): An attribute for enforcing the maximum joint velocity (in radians per second [rad/s] for revolute joints, in metres per second [m/s] for prismatic joints). See safety limits.

### Step 5 – fixing the visual origin

We can now fix the origin of the shoulder\_link visual. (the idea is that 1st we fix the joint and only later the visual of the cylinder)

Modify the <origin> tag inside the <link> tag of the shoulder\_link:

<origin xyz="0 0 0.15" rpy="0 0 0" />

If you start RViz again, you will then see that everything is correctly placed:

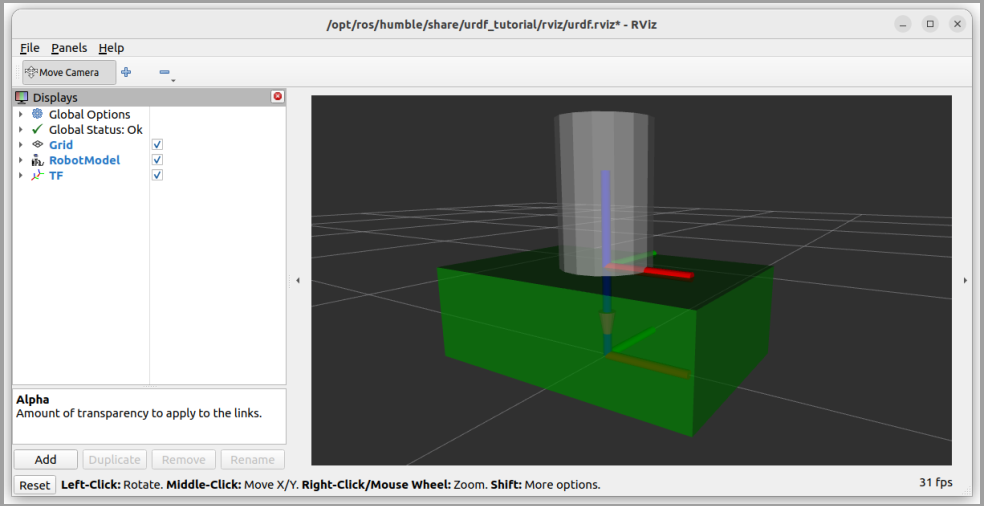


Figure 11.5 – The end of the process for fixing the origins

### Recap – the process to follow every time

## 11.3 Writing a URDF for a mobile robot

### What we want to achieve

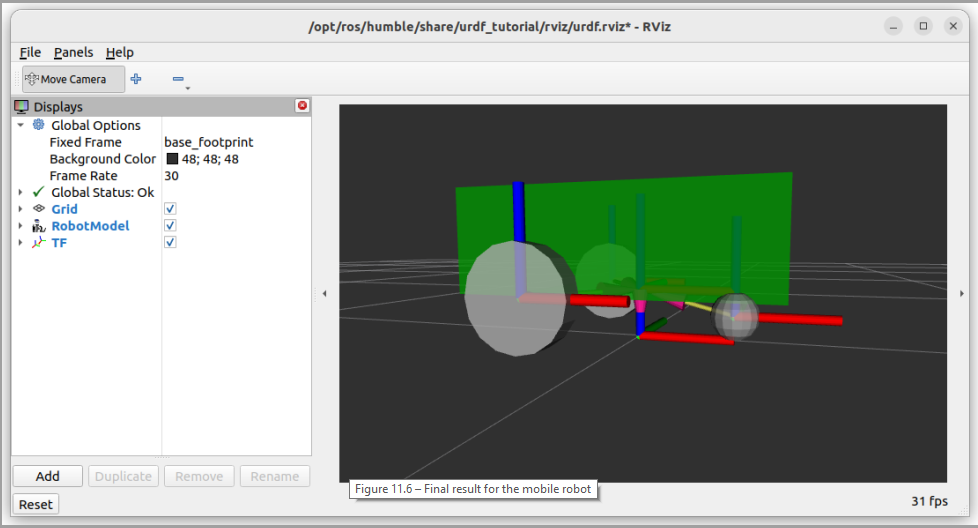


Figure 11.6 – Final result for the mobile robot

### Adding the wheels

#### Right wheel

<link name="right\_wheel\_link">

<visual>

<geometry>

<cylinder radius="0.1" length="0.05" />

</geometry>

<origin xyz="0 0 0" rpy="0 0 0" />

<material name="gray" />

</visual>

</link>

Let’s add a joint:

<joint name="base\_right\_wheel\_joint" type="fixed">

<parent link="base\_link" />

<child link="right\_wheel\_link" />

<origin xyz="0 0 0" rpy="1.57 0 0" />

</joint>

For the wheel/cylinder: We need to have a 90-degree rotation on the *X*-axis (around the red axis). This corresponds to pi/2, or about 1.57 radian (that’s what we have above in rpy=”1.57…”)

Then, the question is: where do we place the right\_wheel\_link frame relative to the base\_link frame (Step 3)? Let’s see for each axis:

* **X translation** (red axis): We want the wheel to be a bit behind, let’s choose a -0.15 m offset.
* **Y translation** (green axis): The wheel should be on the side of the robot.

Let’s modify the <origin> tag inside the <joint> tag:

<origin xyz="-0.15 -0.225 0" rpy="0 0 0" />

We can now easily add the movement (Step 4). As the wheel will continuously rotate (there is no minimum or maximum position), we choose the **continuous** type. We also need to specify the rotation axis. By looking at the robot model in RViz, we can see that we have to pick the Y-axis

<joint name="base\_right\_wheel\_joint" type="continuous">

<parent link="base\_link" />

<child link="right\_wheel\_link" />

<origin xyz="-0.15 -0.225 0" rpy="0 0 0" />

<axis xyz="0 1 0" />

</joint>

You can now start RViz again, disable the wheel visual, and move the new cursor named base\_right\_wheel\_link on the ***Joint State Publisher*** window.

#### Left wheel

<link name="left\_wheel\_link">

<visual>

<geometry>

<cylinder radius="0.1" length="0.05" />

</geometry>

<origin xyz="0 0 0" rpy="1.57 0 0" />

<material name="gray" />

</visual>

</link>

<joint name="base\_left\_wheel\_joint" type="continuous">

<parent link="base\_link" />

<child link="left\_wheel\_link" />

<origin xyz="-0.15 0.225 0" rpy="0 0 0" />

<axis xyz="0 1 0" />

</joint>

#### rpy according to Gemini

In URDF, rpy stands for **Roll**, **Pitch**, **Yaw**. It represents the fixed rotation (also known as extrinsic rotation) applied to a link's coordinate frame relative to its parent link's coordinate frame.

Let's break it down:

* **Roll**: Rotation around the X-axis.
* **Pitch**: Rotation around the Y-axis.
* **Yaw**: Rotation around the Z-axis.

Important Considerations:

Order of Rotation: The order in which these rotations are applied is crucial and fixed in URDF. It's always Roll, then Pitch, then Yaw (often referred to as XYZ or fixed axis rotation). This means the rotations are applied with respect to the original, unrotated coordinate frame of the parent link.

#### Adding the caster wheel

the caster wheel will be represented by a sphere (with a radius of 0.05 m). For the movement, even if the wheel is rotating, this is not a rotation we control with ROS. It is a passive rotation; thus, we will consider the joint as fixed.

<link name="caster\_wheel\_link">

<visual>

<geometry>

<sphere radius="0.05" />

</geometry>

<origin xyz="0 0 0" rpy="0 0 0" />

<material name="gray" />

</visual>

</link>

<joint name="base\_caster\_wheel\_joint" type="fixed">

<parent link="base\_link" />

<child link="caster\_wheel\_link" />

<origin xyz="0 0 0" rpy="0 0 0" />

</joint>

### Adding the caster wheel

Start RViz and disable the visual. From this, let’s see where to place the origin of the caster wheel relative to the origin of the base (Step 3):

* X translation: As the two wheels are on the back of the robot, the caster wheel should be at the front. We can choose for example 0.2 m.
* Y translation: For better stability, we want the caster wheel to be centered, so, 0.
* Z translation: I specified 0.05 as the radius so that the diameter of the caster wheel (0.1 m) corresponds to the radius of the wheel. Thus, in order for the wheels and caster wheels to be aligned on the ground, we need to offset the Z-axis by -0.05 m.

<origin xyz="0.2 0 -0.05" rpy="0 0 0" />

### Extra link – base footprint

it would be nice to have the origin of the robot aligned with the ground where the robot will be.

For that reason, it’s quite common to add a virtual link named base\_footprint, which will be the projection of the base\_link on the ground. We say the link is virtual because it doesn’t contain any visuals;

<link name="base\_footprint" />

Now, we can add a new fixed joint, with the base\_footprint as the parent, and the base\_link as the child:

<joint name="base\_joint" type="fixed">

<parent link="base\_footprint" />

<child link="base\_link" />

<origin xyz="0 0 0.1" rpy="0 0 0" />

</joint>

## 11.4 Improving the URDF with Xacro

*(chapter 28 video)*

The more complex your robot, the bigger the URDF. As you add more links and joints, you will end up having problems scaling your robot model. Also, what we have written so far is not so dynamic: all the values are hardcoded.

Xacro is an additional ROS feature you can use to solve all those issues.

With Xacro, your URDF files will become more dynamic and scalable. All serious ROS 2 projects use Xacro

### Making a URDF file compatible with Xacro

Frist let’s make sure that Xacro is installed (it should already be there with all the previous packages we installed):

$ sudo apt install ros-<distro>-xacro

First, change the file extension. The file is currently named my\_robot.urdf. For Xacro, you will use the .xacro extension. A common practice is to use the .urdf.xacro extension for the main URDF file of your robot, so the file would be named my\_robot.urdf.xacro (the important thing is to have .xacro at the end).

Once you’ve changed the extension, open the file and modify the <robot> tag:

<robot name="my\_robot" xmlns:xacro="http://www.ros.org/wiki/xacro">

### Xacro properties

*(chapter 29 video)*

Variables do not exist in URDF, but you can use them with Xacro. Here, a variable is called a ***property***.

With Xacro properties, we will be able to specify values such as dimensions at the beginning of the file and use those properties inside the <link> and <joint> tags.

Also, it’s important to note that Xacro properties are considered constant variables. After you set their value, you won’t modify them anymore.

To declare and define a Xacro property, you will use the <xacro:property> tag and provide two arguments: name and value

<xacro:property name="base\_length" value="0.6" />

Then, to use a Xacro property, you simply have to write ${property\_name}. You can also do computations. For example, to multiply a value by 2.5, you will write ${property\_name \* 2.5}.

With this information, let’s modify the content inside the base\_link to remove any hardcoded value:

<geometry>

<box size="${base\_length} ${base\_width} ${base\_height}" />

</geometry>

<origin xyz="0 0 ${base\_height / 2.0}" rpy="0 0 0" />

### Xacro macros

*(chapter 32 of video)*

A Xacro macro is the equivalent of a function in programming. With Xacro, a macro works like a template: it’s a piece of XML code that you can reuse with different values (parameters). A macro doesn’t return anything.

Macros are quite useful when you have to duplicate a link or a joint several times. Imagine a robot with four cameras. You can create a macro for the camera link, and then call the macro instead of re-writing the same code four times.

To create a macro, you will use the **<xacro:macro>** tag and give a name as well as a list of params. You can specify zero, or as many parameters as you want—just separate them with a space.

Eg.

<xacro:macro name="wheel\_link" params="prefix">

<link name="${prefix}\_wheel\_link">

<visual>

<geometry>

<cylinder radius="${wheel\_radius}" length="${wheel\_length}" />

</geometry>

<origin xyz="0 0 0" rpy="${pi / 2.0} 0 0" />

<material name="gray" />

</visual>

</link>

</xacro:macro>

This piece of code won’t do anything by itself. We need to call it, just like you would call a function:

<xacro:wheel\_link prefix="right" />

<xacro:wheel\_link prefix="left" />

### Including a Xacro file in another file

*(Chapter 33 video)*

Thanks to Xacro, you can split your URDF into several files.

Coming back to our URDF, let’s split the file into three:

* common\_properties.xacro: This will contain the material tags and other properties that could apply to any part of our robotics application
* mobile\_base.xacro: This file will contain the properties, macros, links, and joints that are specific to the mobile base
* my\_robot.urdf.xacro: In this main file, we include the two previous files

To include a Xacro file inside another file, you will write a <xacro:include> tag and provide the path to the file with the filename attribute.

my\_robot.urdf.xacro:

<?xml version="1.0"?>

<robot name="my\_robot" xmlns:xacro="http://www.ros.org/wiki/xacro">

<xacro:include filename="common\_properties.xacro" />

<xacro:include filename="mobile\_base.xacro" />

</robot>

Let’s start with common\_properties.xacro.

<?xml version="1.0"?>

<robot xmlns:xacro="http://www.ros.org/wiki/xacro">

<material name="green">

<color rgba="0 0.6 0 1" />

</material>

<material name="gray">

<color rgba="0.7 0.7 0.7 1" />

</material>

</robot>

All Xacro files must have this code, with a <robot> tag containing the xmlns:xacro attribute.

Note: Don’t add the name attribute in the <robot> tag. This attribute will only be added once in

the main Xacro file.

mobile\_base.xacro:

<?xml version="1.0"?>

<robot xmlns:xacro="http://www.ros.org/wiki/xacro">

<xacro:property name="base\_length" value="0.6" />

<xacro:property name="base\_width" value="0.4" />

<xacro:property name="base\_height" value="0.2" />

<xacro:property name="wheel\_radius" value="0.1" />

<xacro:property name="wheel\_length" value="0.05" />

<link name="base\_footprint" />

<link name="base\_link">

<visual>

...

<joint name="base\_caster\_wheel\_joint" type="fixed">

<parent link="base\_link" />

<child link="caster\_wheel\_link" />

<origin xyz="${base\_length / 3.0} 0 ${-wheel\_radius / 2.0}" rpy="0 0 0" />

</joint>

</robot>

## 11.5 Summary

# Chapter 12 - Publishing TFs and Packaging the URDF

## 12.1 Understanding how to publish TFs with our URDF

### The robot\_state\_publisher node

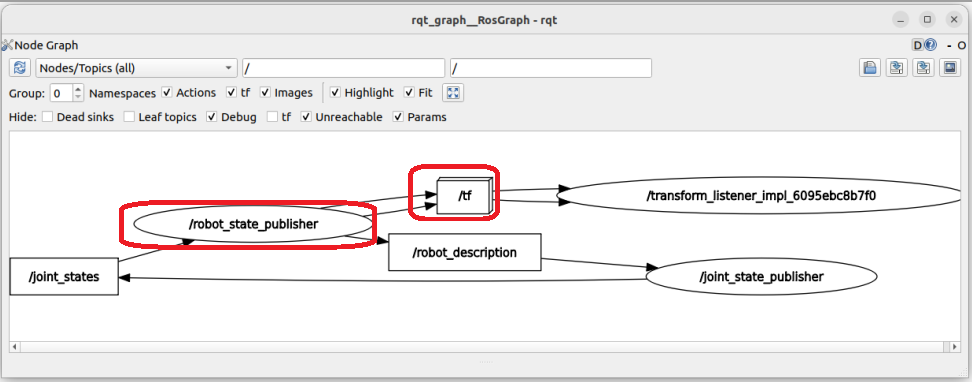


Figure 12.1 – Nodes running with urdf\_tutorial

Now, what is publishing on the /tf topic? As you can see, there is a node called /robot\_state\_publisher.

Most of the time, you won’t have to publish any TF by yourself, as this will be handled by the robot\_state\_publisher node.

### What is the robot\_state\_publisher in ros (According to Gemini)?

In the Robot Operating System (ROS), the robot\_state\_publisher is a crucial package that plays a key role in visualizing and understanding the state of a robot. Here's a breakdown of its purpose:

* **Publishing Robot State to tf:**
  + The primary function of robot\_state\_publisher is to publish the state of a robot to the tf (transform) system. tf is a ROS package that allows you to keep track of multiple coordinate frames over time.
  + Essentially, it takes the joint angles of a robot as input and calculates the 3D poses of the robot's links.
  + This information is then broadcasted, making it available to other ROS components that rely on tf.
* **Kinematic Tree Model:**
  + The robot\_state\_publisher utilizes a kinematic tree model of the robot, typically defined in a URDF (Unified Robot Description Format) file.
  + This model provides the structural information about the robot, including the links and joints that make up its body.
* **Key Functionality:**
  + It allows for the translation of joint state information, to the transformation of the robots links in 3D space. This is vital for tools like Rviz, which is a 3D visualizer for ROS.
  + It publishes static transforms, and dynamic transforms. Static transforms are published once, and dynamic transforms are published as joint states change.

In simpler terms, the robot\_state\_publisher helps ROS understand where each part of a robot is in 3D space, which is essential for tasks like visualization, navigation, and manipulation.

### Inputs for the robot\_state\_publisher

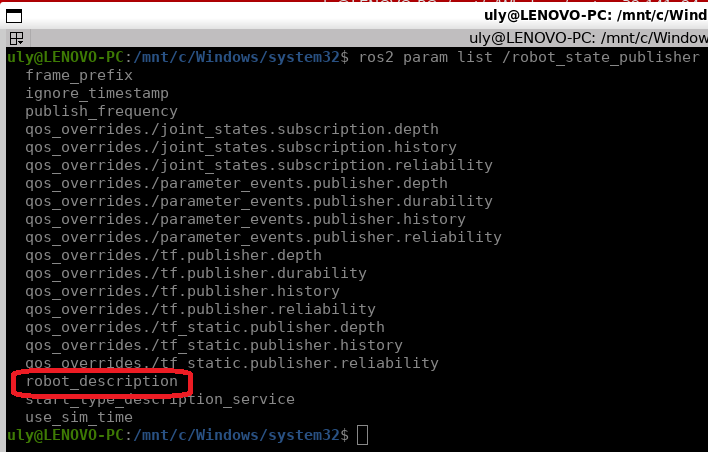
There are two things you need to provide for the robot\_state\_publisher node to work correctly:

URDF and joint states. Let’s start with the first one.

#### URDF as a parameter

list all parameters for the robot\_state\_publisher node:

$ ros2 param list /robot\_state\_publisher

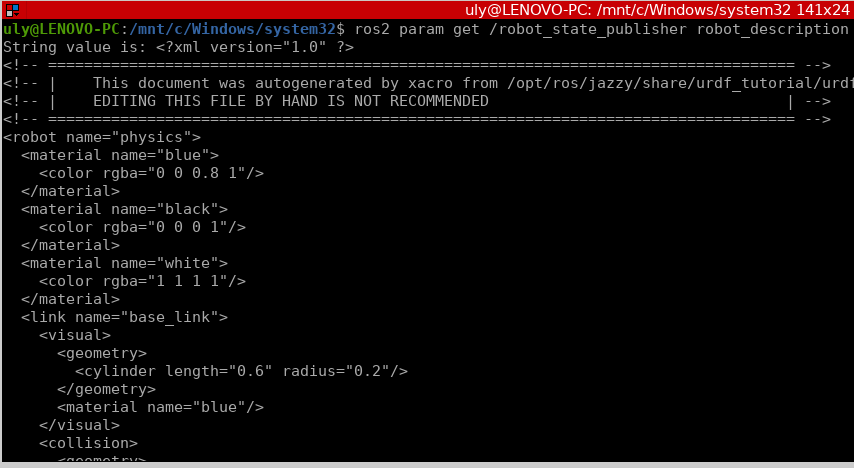


You will see quite a few, but the one that we care about here is named robot\_description.

Then, you can read the value from this parameter:

$ ros2 param get /robot\_state\_publisher robot\_description

With this, you will see the entire URDF in the terminal



So, when you start the robot\_state\_publisher node, you will need to give the URDF inside a parameter named robot\_description.

That’s it for the first input; let’s see the second one.

#### Joint states topic

In order to publish the TFs, the robot\_state\_publisher node will need the URDF, but also the current state for each joint.

You can see the /joint\_states topic in rqt\_graph in Figure 12.1. This topic contains what you would read from encoders or control feedback in a real robot. For example, if you have some wheels, you will get to know the speed and/or position of those wheels. You will feed this into the /joint\_states topic. If you have a robotic arm, you usually have encoders on each axis reading the current position for the axis.

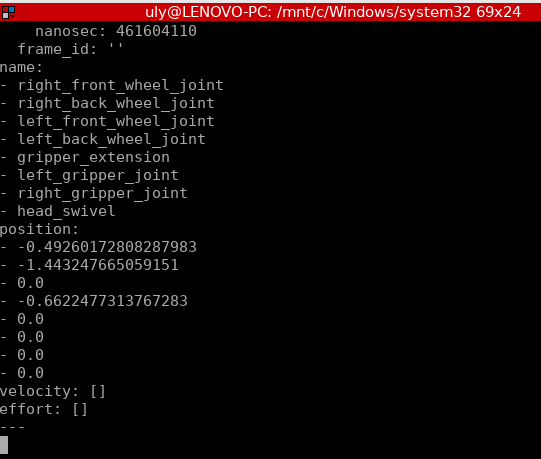
When we simulate the robot with Gazebo, we will use a plugin that automatically publishes the joint states. In fact, either in simulation mode or for a real robot, you will usually have nodes doing this for you.

For now, as we don’t have any real robot or Gazebo simulation, we will use the joint\_state\_publisher node (with the Joint State Publisher window), which will publish whatever values we select on the cursors. For example, if you select 1.0 radian for base\_right\_wheel\_joint, then 1.0 will be published on the /joint\_states topic for that joint and will be received and used by robot\_state\_publisher.

Here, it’s important to clarify the difference between joint states and TFs. A joint state is simply the *current state* of a joint. For example, for a wheel: what is the current velocity? It doesn’t specify anything about the relationship between joints, nor where they are located relative to each other—this is what a TF is.

Ulisses’ Note: below you have a console print-screen of the joint states:

$ ros2 topic echo /joint\_states



### Recap – how to publish TFs

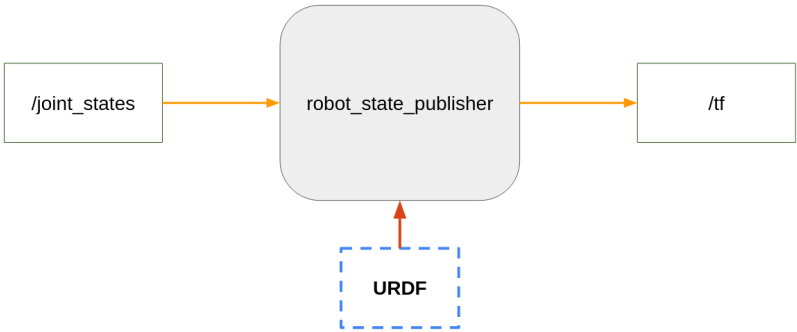


Figure 12.2 – The required node and inputs to publish TFs

The root\_state\_publisher receives the URDF and the joint\_states to create/publish the tf.

## 12.2 Starting all nodes from the terminal

### Publishing the TFs from the terminal

Let’s publish the TFs. For that, we will open two terminals.

In the first one, start the robot\_state\_publisher node. The package and executable names for this node are identical. To provide the robot\_description parameter, you will have to use this syntax: "$(xacro <path\_to\_urdf>)". (this parameter will contain the urdf file)

In Terminal 1, run the following command:

$ ros2 run robot\_state\_publisher robot\_state\_publisher --ros-args -p robot\_description:="$(xacro /home/<user>/my\_robot.urdf.xacro)"

(We give 1st the name of the package and then the name of the node, that have the same name “robot\_state\_publisher”)

[robot\_state\_publisher]: got segment base\_footprint

[robot\_state\_publisher]: got segment base\_link

[robot\_state\_publisher]: got segment caster\_wheel\_link

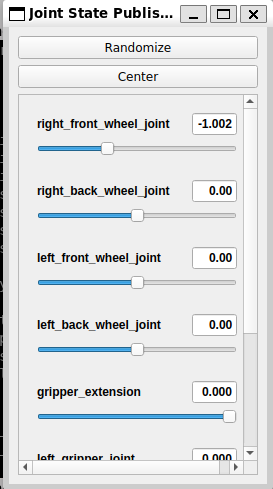
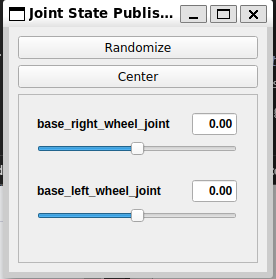
[robot\_state\_publisher]: got segment left\_wheel\_link

[robot\_state\_publisher]: got segment right\_wheel\_link

In Terminal 2, run the following command:

$ ros2 run joint\_state\_publisher\_gui joint\_state\_publisher\_gui

This will open the Joint State Publisher window

 or 

### Visualizing the robot model in RViz

#### Starting and configuring RViz

in Terminal 3, run the following command:

$ ros2 run rviz2 rviz2

This will open an empty RViz

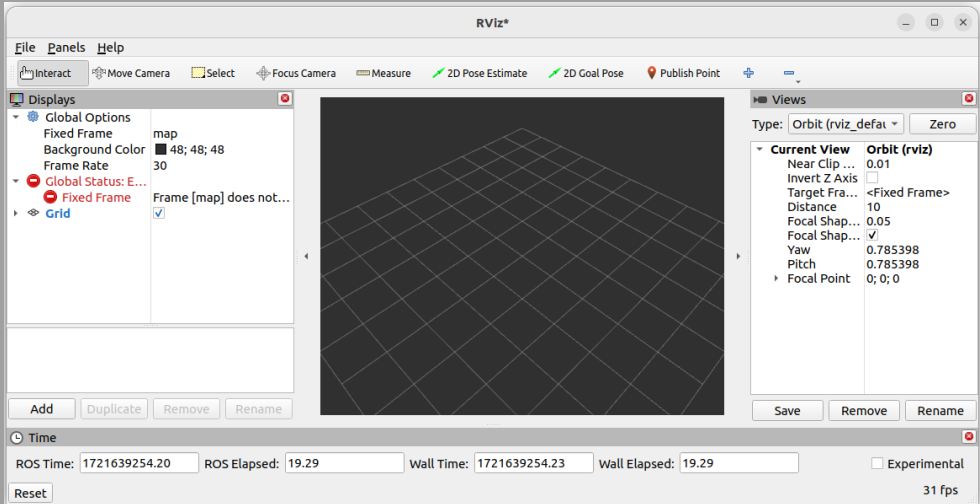
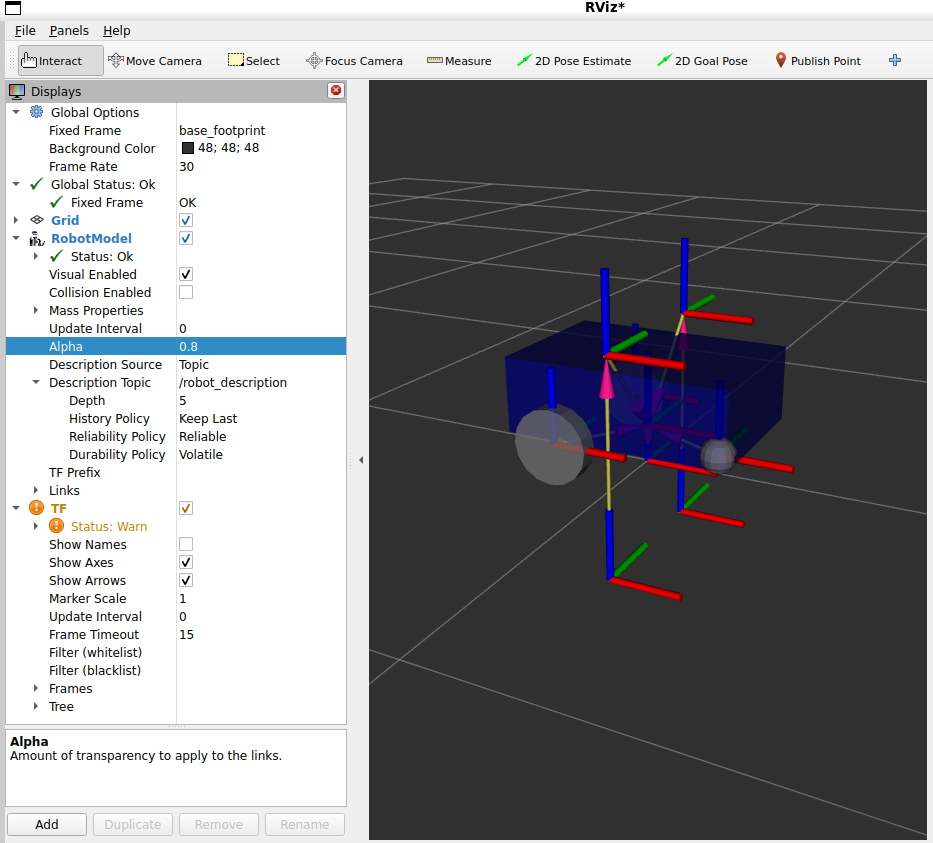


Figure 12.3 – RViz with no robot model and some errors

We need to do a bit of configuration to correctly visualize the robot model and the TFs. Then, we will be able to save this configuration and reuse it the next time we start RViz.

Follow these steps to configure RViz:

1. In the left menu, ***Global Options | Fixed Frame***, change from map to base\_footprint. After that, Global Status: Error should change to Global Status: OK.
2. Click on the ***Add*** button on the left, scroll down, and double-click on RobotModel. You will have a new menu on the left side of RViz.
3. Open this new ***RobotModel*** menu, find ***Description Topic***, and click on the empty space on the right side of the menu (this one is a bit tricky to find). You should see a drop-down menu; here, select /robot\_description. After this, the robot model should appear on the screen.
4. Click on the ***Add*** button again, scroll down, and double-click on ***TF***. This will open a new menu, and you will see the TFs appear on the screen.
5. If you want to see through the model, like we did before, open ***RobotModel***, and reduce the ***Alpha*** (transparency) value from 1 to 0.8, for example.
6. You can remove the extra menus on the right (***Views***) and at the bottom (***Time***) to get more space for the robot



#### Saving the RViz configuration

Click on File | Save Config As. Let’s name the file urdf\_config.rviz (for these files, use the .rviz extension),

Then, when you start RViz again, you can add an extra -d argument with the path to the configuration file:

$ ros2 run rviz2 rviz2 -d /home/<user>/urdf\_config.rviz

This will start RViz exactly like you saved it: same menus, same view, same zoom, and so on.

## 12.3 Creating a package to install the URDF

### Adding a new workspace

(chapter 22 of video)

Let’s start by creating the folder:

$ mkdir ~/my\_robot\_ws

Then, inside this workspace, create a src directory:

$ mkdir ~/my\_robot\_ws/src

In case the .bashrc file was pointing to/sourcing another workspace, we should comment that line by putting # at the beginiing.

let’s build and source our new workspace

$ cd ~/my\_robot\_ws/

$ colcon build

And now we would need to add the new workspace to the .bashrc file:

source /opt/ros/jazzy/setup.bash

#source ~/ros2\_ws/install/setup.bash

source ~/my\_robot\_ws/install/setup.bash

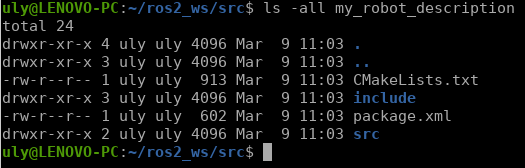
### Creating a \_description package

Now that we have this new empty workspace correctly configured and sourced, let’s add a new package inside.

To name the package, we use the robot’s name, followed by “\_description”. This is a common convention used by lots of ROS developers.

$ cd ~/my\_robot\_ws/src/

$ ros2 pkg create my\_robot\_description --build-type ament\_cmake



For now, this package is a standard C++ ROS 2 package. However, we won’t write any nodes inside. We just need the package to install our robot model. Thus, you can remove the src and include directories:

Open the workspace with an IDE. If you’re using VS code, run the following:

$ cd ~/my\_robot\_ws/src/

$ code .

To clean things a bit more, we will simplify the CMakeLists.txt file. Remove the comments after the find\_package(ament\_cmake REQUIRED) line, and, as we don’t need that now, remove the if(BUILD\_TESTING) block. Make sure you keep the ament\_package() instruction, which should be the last line of the file.

### Installing the URDF and other files

#### Installing Xacro and URDF files

To install our Xacro and URDF files, go inside the my\_robot\_description package and create a new folder named urdf:

$ cd ~/my\_robot\_ws/src/my\_robot\_description/

$ mkdir urdf

You can now move all three Xacro files inside this urdf folder

Now, open CMakeLists.txt, and add those instructions to install the urdf folder:

find\_package(ament\_cmake REQUIRED)

install(

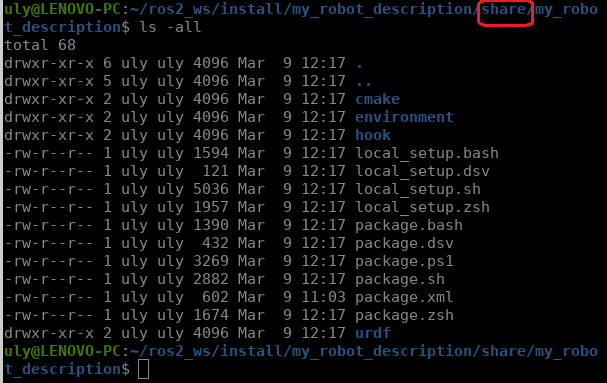
DIRECTORY urdf

DESTINATION share/${PROJECT\_NAME}/

)

ament\_package()

This will install the urdf folder inside a share directory when you build the package. It will allow any package from the workspace to find the URDF for your robot.



#### Installing custom meshes

*(Chapter 35 video)*

A mesh could be replacing the built-in <box> or <cylinder>. A mesh could come from *Blender* or *SolidWorks*.

in my\_robot\_description, you would create a new folder named meshes:

$ cd ~/my\_robot\_ws/src/my\_robot\_description/

$ mkdir meshes

In this folder, you would add all the .stl and .dae files you want to use in your URDF. Then, let’s say you have added a file named base.stl. In the URDF, you will use this syntax to include it:

<mesh filename="file://$(find my\_robot\_description)/meshes/base.stl"/>)

To install the meshes folder, you also have to add an instruction in CMakeLists.txt. As we already added the install() block previously, you just need to add the name of the new folder you want to install:

install(

DIRECTORY urdf meshes

DESTINATION share/${PROJECT\_NAME}/

)

#### Installing the RViz configuration

Let’s install the RViz configuration we previously saved. This way, when we start RViz later on from a launch file, we can use this configuration directly from the package.

Create an rviz folder inside the package:

$ cd ~/my\_robot\_ws/src/my\_robot\_description/

$ mkdir rviz

Move the urdf\_config.rviz file into this new rviz folder.

## 12.4 Writing a launch file to publish TFs and visualize the robot

*(chapter 23 of the video)*

The my\_robot\_description package is finished, but we will add a launch file so we can start all the nodes and parameters that we discovered at the beginning of this chapter. This way, we can publish TFs and visualize the robot in RViz. This will also be a good practice exercise on launch files, and we will reuse part of the code in the next chapter when we build the Gazebo simulation for the robot.

### The XML launch file

Before writing any launch file, we first need to create a launch folder, where we will put all our launch files for the my\_robot\_description package.

#### Creating and installing a launch folder

$ cd ~/my\_robot\_ws/src/my\_robot\_description/

$ mkdir launch

Then, to specify the instructions to install this folder, it’s as easy as adding the folder name inside CMakeLists.txt (add "launch" at the end of the DIRECTORY line in the file):

install(

DIRECTORY urdf meshes rviz launch

DESTINATION share/${PROJECT\_NAME}/

)

This file will be used by the colcon build to find out multiple things including which directories will be copied to the install directory.

Once this is done, create a new launch file inside the folder. We will name it display.launch.xml:

$ cd launch/

$ touch display.launch.xml

#### Writing the launch file

In the launch file, first open and close a <launch> tag:

<launch>

</launch>

Now, make sure to write all the following lines inside this <launch> tag.

to add (constant) variables in a launch file use the <let> tag with two arguments: name and value.

<let name="urdf\_path" value="$(find-pkg-share my\_robot\_description)/urdf/my\_robot.urdf.xacro" />

<let name="rviz\_config\_path" value="$(find-pkg-share my\_robot\_description)/rviz/urdf\_config.rviz" />

Then, to use a variable, you can write $(var name).

Note: To find a package, you use find-pkg-share in a launch file

*(Chapter 34 Video)*

Now, let’s start all the nodes we need, one by one. Here is the code for the robot\_state\_publisher node:

<node pkg="robot\_state\_publisher" exec="robot\_state\_publisher">

<param name="robot\_description"

value="$(command 'xacro $(var urdf\_path)')" />

</node>

we use the same values as in the command we previously ran in the terminal. To specify a command to run in an XML launch file, you can use $(command '...').

Note from video: The “xacro” utility/command above will get the xacro file and generate a standard urdf file. So, xacro is like a pre-processor.

Next, we can start the joint\_state\_publisher node. Here, we use the executable with the \_gui suffix to get a graphical window with cursors to move the joints:

<node pkg="joint\_state\_publisher\_gui" exec="joint\_state\_publisher\_gui" />

Let’s finish with the RViz node:

<node pkg="rviz2" exec="rviz2" args="-d $(var rviz\_config\_path)" />

In this node, we provide the saved RViz configuration file, using the -d option.

The full display.launc.xml file would look like:

<launch>

<let name="urdf\_path"

value="$(find-pkg-share my\_robot\_description)/urdf/my\_robot.urdf.xacro" />

<let name="rviz\_config\_path"

value="$(find-pkg-share my\_robot\_description)/rviz/urdf\_config.rviz" />

<node pkg="robot\_state\_publisher" exec="robot\_state\_publisher">

<param name="robot\_description"

value="$(command 'xacro $(var urdf\_path)')" />

</node>

<node pkg="joint\_state\_publisher\_gui" exec="joint\_state\_publisher\_gui" />

<node pkg="rviz2" exec="rviz2" args="-d $(var rviz\_config\_path)" />

</launch>

#### Starting the launch file

The launch file is now complete. We can build the workspace to install all the files and folders that we’ve added:

$ cd ~/my\_robot\_ws/

$ colcon build --packages-select my\_robot\_description

Then, don’t forget to source the workspace (source install/setup.bash), and you can start your new launch file:

$ ros2 launch my\_robot\_description display.launch.xml

### The Python launch file

## 12.5 Summary

# Chapter 13 - Simulating a Robot in Gazebo

## 13.1 Technical requirements

## 13.2 How Gazebo works

### Clarifying – Gazebo versus RViz

RViz is not a simulation tool. You don’t simulate anything; you only visualize what already exists.

Gazebo is a simulation tool. It will simulate gravity and the real physical properties of the robot. It also has some control plugins so that you can simulate the hardware control, and even publish the joint states and TFs for your robot.

### Starting Gazebo

for Gazebo, install this additional package:

$ sudo apt install ros-jazzy-ros-gz

Note: Since we are using Ubuntu 24.04, this will install Gazebo Harmonic.

to start Gazebo:

$ gz sim

You will be taken to a Gazebo quick-start menu. There, you can click on ***Empty*** to load an empty world.

When running the Gazebo command, you can also directly specify the world you want to launch. World description files in Gazebo use the SDF format (a file with a .sdf extension)

$ gz sim empty.sdf

At the bottom left of the screen, you will see a play button. Click on it to start the simulation.

**Note**

Gazebo can be quite buggy, so don’t be surprised if it crashes at some point—even on a powerful computer. If you can’t close Gazebo properly (with Ctrl + C in the terminal), you may have some trouble when starting it again. In this case, you can try to stop all Gazebo processes that might still be running in the background. To do that, run ps aux | grep gz to find all related processes. You will find a pid with four numbers for each gz process (if any). To stop a process, run kill –9 <pid>. If nothing works, the best thing to do is to restart your computer.

at the top of the gazebo screen, you can click on the different shapes and add them to the empty space. Add a box into the space. Find the translation mode and rotation mode. Move the box around (especially on the *z* axis) and see what happens. If you lift the box up, then you should see the box falling down on the floor *(after pressing the small orange run button at the bottom left)*.

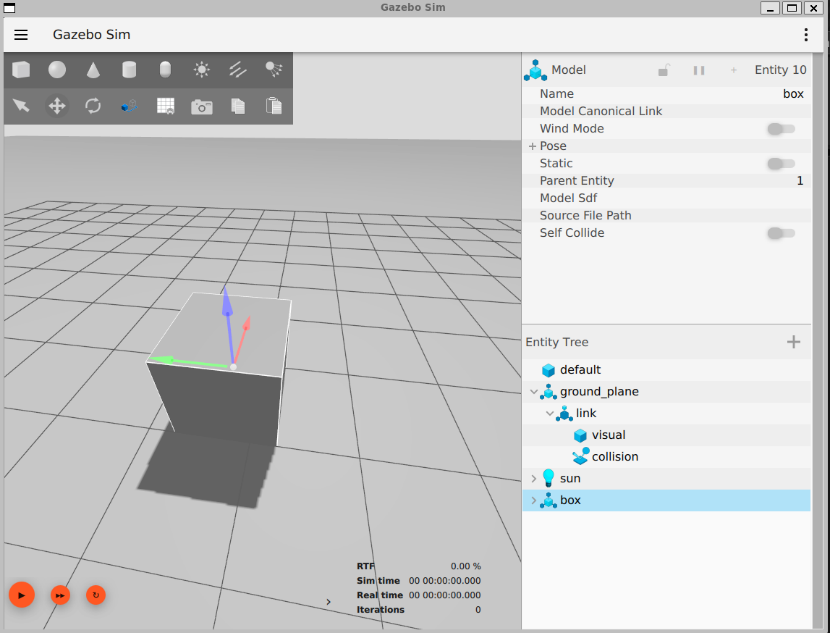


Figure 13.1: The Gazebo simulator with a box in an empty world

Gazebo also uses topics and services, but those are not the same as ROS 2 topics and services.

For example, you can list all Gazebo topics with this command:

$ gz topic -l

### How Gazebo works with ROS 2

We will now explore how Gazebo and ROS 2 can be connected.

First of all, you can start Gazebo using a ROS 2 launch file from the ros\_gz\_sim package. This will be more practical for us because when we write our own launch file, we can include this one:

$ ros2 launch ros\_gz\_sim gz\_sim.launch.py

This will start Gazebo the same way we did with the gz sim command. You can also specify the world to launch with the gz\_args argument:

$ ros2 launch ros\_gz\_sim gz\_sim.launch.py gz\_args:=empty.sdf

However, even if we started Gazebo from a ROS 2 launch file, Gazebo is still independent.

*(See also: Chapter 40 Video – ROS 2 for Beginners Level 2 - Udemy)*

To connect Gazebo and ROS 2 topics (or services), you need to create a bridge between them. The **ros\_gz\_bridge** package does that for us, so we will use this package. We will only need to provide some configuration to specify which topics we want to bridge

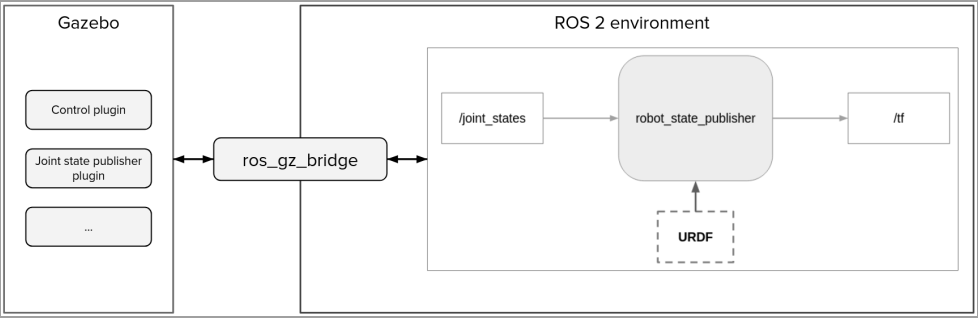


Figure 13.2: Connecting Gazebo and ROS 2 with ros\_gz\_bridge

In Figure 13.2, you can see the following:

* On the right, our current ROS 2 application with the robot\_state\_publisher node, publishing on the /tf topic.
* On the left, Gazebo. Inside Gazebo, we will add plugins (also called systems) to simulate the hardware behavior of the robot. For example, you could have one plugin to control the two wheels, and one plugin to publish the joint states for the wheels. This is what we will implement in this chapter.
* Then, to make everything work together, we will use the **ros\_gz\_bridge** package. With the joint state example, the Gazebo Joint state publisher plugin will publish the joint states with a Gazebo topic. Using ros\_gz\_bridge, we will match this topic with the ROS 2 /joint\_states topic.

The important thing to understand here is that Gazebo and ROS 2 exist in two different environments, but you can make them work together. Your ROS 2 application will be the same, whether you work on a Gazebo simulation or a real robot. If you work on a real robot, then you would directly control the wheels and get the joint state data from encoders. With Gazebo, you use plugins to simulate the hardware.

Now, here are the steps we will take in the following sections to create the Gazebo simulation for our robot:

1. Adapt the URDF for Gazebo. For a robot to work on Gazebo, we first need to provide inertial and collision properties in the URDF.
2. Once the URDF is correct, we will start Gazebo and spawn the URDF in it. At this point, we will also create a package with a launch file.
3. We will then add some plugins (systems) to control the robot, using the ros\_gz\_bridge package to make those plugins communicate with our ROS 2 application.

## 13.3 Adapting the URDF for Gazebo

*(Chapter 41, 42 Video – ROS 2 for Beginners Level 2 - Udemy)*

Gazebo URDF needs inertial and collision properties. For each link of the robot that represents a physical part, we will add an <inertial> tag and a <collision> tag.

### Inertial tags

A URDF without inertial properties won’t load in Gazebo. Thus, this is the first thing you need to add.

An <inertial> tag will contain a few elements, including a 3x3 matrix representing an inertia tensor.

So, currently, our URDF is split into three files. We will add some code to those files:

* common\_properties.xacro: Here, we will add some macros to specify the inertial properties for a box, a cylinder, and a sphere. This way, we only need to write the inertial formulas once for those shapes, and you can reuse them in any of your projects.
* mobile\_robot.xacro: Inside each link representing a physical part, we will use the corresponding inertial macro we defined previously.
* my\_robot.urdf.xacro: Nothing changes here; we still import the two previous files.

#### What do we write inside an <inertial> tag?

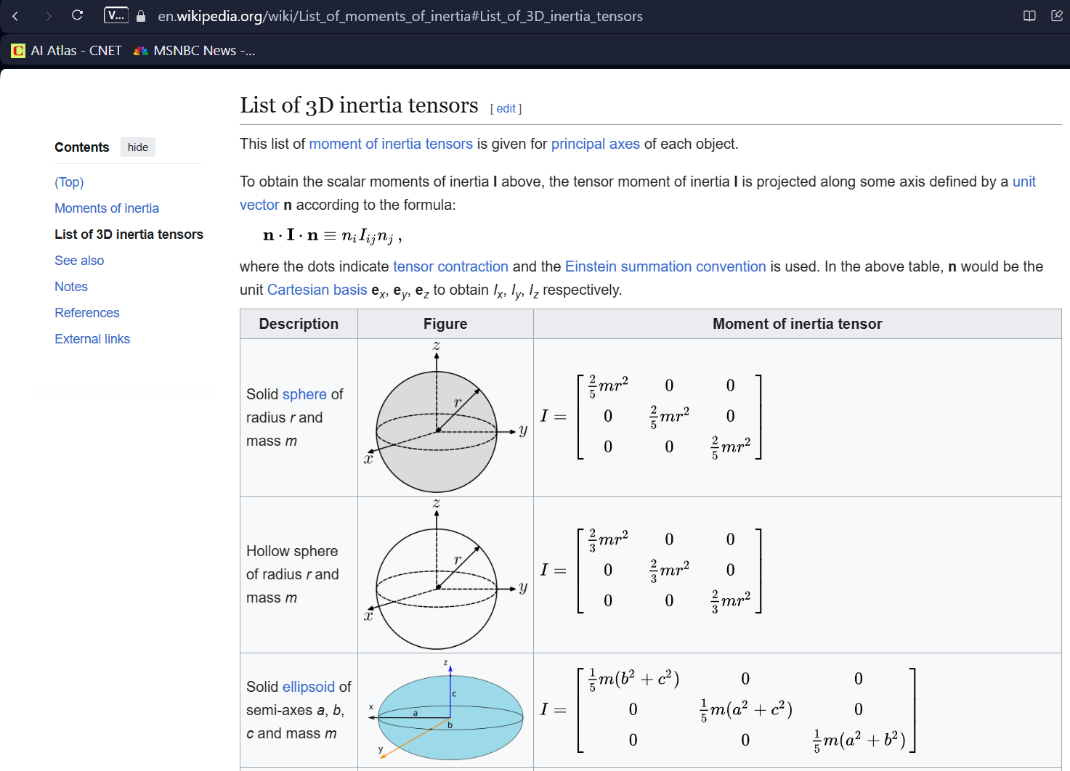
We will create three Xacro macros containing an <inertial> tag—one for a box, one for a cylinder, and one for a sphere. Inside a URDF <inertial> tag, you will need to provide:

* The mass of the element (in kg).
* The origin for the inertia (in meters and radians).
* The nine elements of the inertia tensor, or matrix (in kg per sqm). Since the matrix is symmetrical, we only need six elements—ixx, ixy, ixz, iyy, iyz, and izz (for example, ixy and iyx are the same, so we omit the second one).

Now, how do we compute the inertia matrix? This is usually the hardest part when writing the <inertial> tags.

If you are designing your robot with CAD software—for example, with **SolidWorks**—then you can export each property (for inertia) directly from the software and add them to your URDF. As we don’t have this software, we will need to make the computation ourselves.

Fortunately, there are some helpful resources on the internet. You can find a list of moments of inertia on Wikipedia: <https://en.wikipedia.org/wiki/List_of_moments_of_inertia>.



There, you can also find the moment of inertia for each simple shape that we have, as well as a list of 3D inertia tensors, which are basically the matrices we need for the URDF. One thing to note is that the matrices only have three non-zero components—ixx, iyy, and izz. All the other components are set to 0.

Ulisses’ Note: It could be useful to also read the official documentation about this (Adding Physical and Collision Properties to URDF):

<http://wiki.ros.org/urdf/Tutorials/Adding%20Physical%20and%20Collision%20Properties%20to%20a%20URDF%20Model>

#### Adding inertial macros for basic shapes

As the inertial macros for basic shapes could be used by any robot, we will add all macros in the common\_properties.xacro file. This way, if you want to create another URDF for another robot, you can just reuse this Xacro file.

The first macro will be for a box inertia. Now, if you look at the preceding Wikipedia link, things could be a bit confusing, as they use width, depth, and height (w, d, and h). In ROS 2, we have specified the length, width, and height for the x, y, and z dimensions. Which one corresponds to which?

Here is what we will use (on the left, the Wikipedia value, and on the right, the ROS 2 value):

* **w**: the x dimension
* **d**: the y dimension
* **h**: the z dimension (this is also the axis pointing up)

Here is the <inertial> tag for a box. You can add this after the <material> tags, inside the <robot> tag:

<xacro:macro name="box\_inertia" params="m x y z o\_xyz o\_rpy">

<inertial>

<mass value="${m}" />

<origin xyz="${o\_xyz}" rpy="${o\_rpy}" />

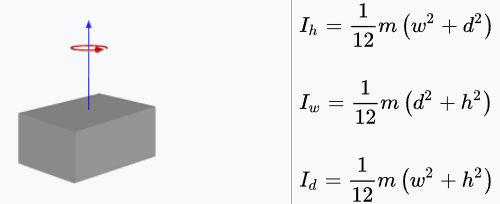
<inertia ixx="${(m/12) \* (z\*z + y\*y)}" ixy="0" ixz="0"

iyy="${(m/12) \* (x\*x + z\*z)}" iyz="0"

izz="${(m/12) \* (x\*x + y\*y)}" />

</inertial>

</xacro:macro>



Let’s now write the macro for a cylinder. This one is a bit easier. We have two components—radius and height. This will correspond to the radius and length we defined in the URDF:

<xacro:macro name="cylinder\_inertia" params="m r l o\_xyz o\_rpy">

<inertial>

<mass value="${m}" />

<origin xyz="${o\_xyz}" rpy="${o\_rpy}" />

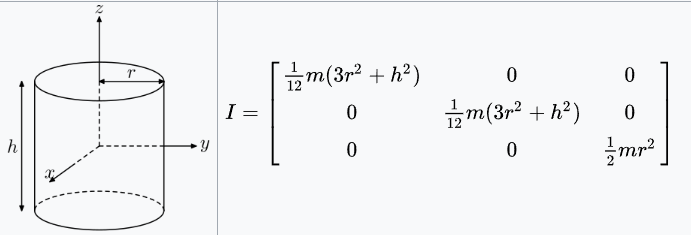
<inertia ixx="${(m/12) \* (3\*r\*r + l\*l)}" ixy="0" ixz="0"

iyy="${(m/12) \* (3\*r\*r + l\*l)}" iyz="0"

izz="${(m/2) \* (r\*r)}" />

</inertial>

</xacro:macro>



Finally, we can write the macro for a (solid) sphere. This is the easiest, and we only have one component — the sphere radius:

<xacro:macro name="sphere\_inertia" params="m r o\_xyz o\_rpy">

<inertial>

<mass value="${m}" />

<origin xyz="${o\_xyz}" rpy="${o\_rpy}" />

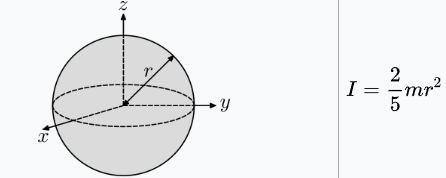
<inertia ixx="${(2/5) \* m \* r \* r}" ixy="0" ixz="0"

iyy="${(2/5) \* m \* r \* r}" iyz="0"

izz="${(2/5) \* m \* r \* r}" />

</inertial>

</xacro:macro>



#### Including the inertial macros in the links

We can now use those macros to provide the inertial property for each link of the robot.

As the base\_footprint doesn’t represent a physical part (it’s what we call a virtual link), it won’t have an inertia. For all the other links (base, right wheel, left wheel, and sphere), we will use the inertial macros.

Open the mobile\_base.xacro file, which is where we will continue with the code.

Now, to add the inertial property for a link, you need to add an <inertial> tag inside the <link> tag. To add this tag, we will use the macros we previously created.

Let’s start with base\_link. Inside the <link name="base\_link"></link> tag, and after the <visual> tag, add the box\_inertia macro:

<xacro:box\_inertia m="5.0" x="${base\_length}" y="${base\_width}" z="${base\_height}"

o\_xyz = "0 0 ${base\_height / 2.0}" o\_rpy="0 0 0" />

We specify all the parameters required for the macro:

* **Mass**: Here, we decided that the box would be 5.0 kg.
* **Box properties**: The x, y, and z dimensions (we have created the macro so that we can use the ROS 2 axis system convention).
* **Inertia origin**: The inertia is related to the box itself, so if the box has an offset relative to the joint origin, you need to take this offset into account. Basically, you can use the same values you wrote in the visual origin.

Now, add the inertia for the two wheels. You will add the cylinder\_inertia macro inside the wheel\_link macro. This inertia macro will apply to both wheels:

<xacro:cylinder\_inertia m="1.0" r="${wheel\_radius}" l="${wheel\_length}"

o\_xyz="0 0 0" o\_rpy="${pi / 2.0} 0 0" />

Here are the parameters we specify:

* **Mass**: Set the mass for each wheel to 1.0 kg.
* **Cylinder properties**: The radius and length of the cylinder.
* **Inertia origin**: The visual is centered around the link origin, so we don’t need to add any offset. However, to match the image on the Wikipedia page, we have defined the macro for the inertia of a cylinder with the rotation axis as the z axis. The wheels’ visual has been shifted by 90° on the x axis so that the rotation axis becomes the y axis. Here, we provide the same rotation for the origin. Basically, once again, you can use the same values you wrote in the visual origin.

The full weel\_link would be (*not in the book*):

<xacro:macro name="wheel\_link" params="prefix">

<link name="${prefix}\_wheel\_link">

<visual>

<geometry>

<cylinder radius="${wheel\_radius}" length="${wheel\_length}" />

</geometry>

<origin xyz="0 0 0" rpy="${pi / 2.0} 0 0" />

<material name="grey" />

</visual>

<xacro:cylinder\_inertia m="1.0" r="${wheel\_radius}" h="${wheel\_length}"

xyz="0 0 0" rpy="${pi / 2.0} 0 0" />

</link>

</xacro:macro>

Finally, we add the sphere\_inertia macro for caster\_wheel\_link:

<xacro:sphere\_inertia m="0.5" r="${wheel\_radius / 2.0}"

o\_xyz="0 0 0" o\_rpy="0 0 0" />

#### Validating inertia with RViz

To make sure that the inertial property is correct for each link, you can use RViz.

start the launch file:

$ ros2 launch my\_robot\_description display.launch.xml

To see the inertia, first disable the visual. On the left menu, open **RobotModel** and uncheck **Visual Enabled**. Then, still inside RobotModel, open **Mass Properties** and check the **Inertia** box. You should see something like this:

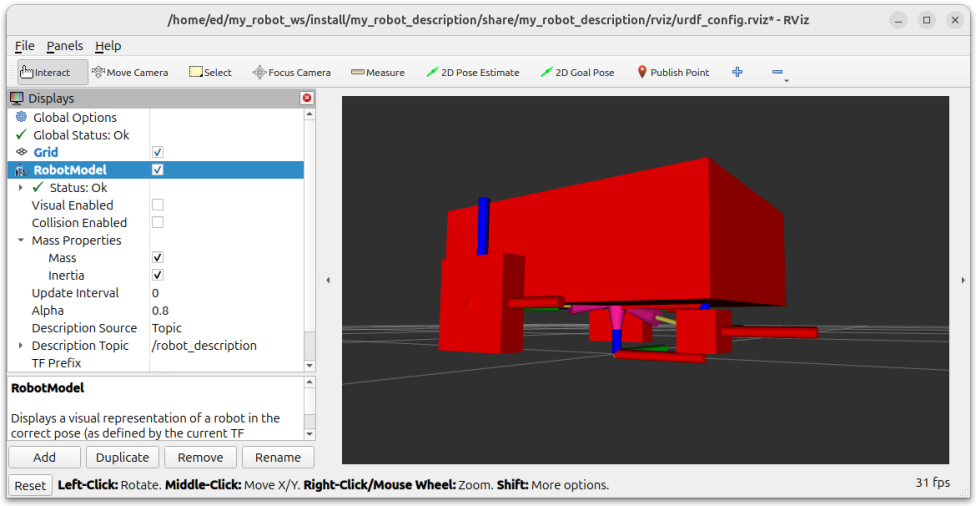


Figure 13.3: Visualizing inertia in Rviz

With this view, you can easily spot errors. For example, if the offset for the base\_link inertia is not right, then the box will not be correctly placed. Another common error will be in the rotation of the wheel inertia. In the preceding figure, you can see that the inertia box is more or less on top of the wheel, correctly orientated.

### Collision tags

*(Chapter 45 Video – ROS 2 for Beginners Level 2 - Udemy)*

Inside each link of our URDF, we have a <visual> tag to see the link and an <inertial> tag to describe the physical properties for Gazebo.

However, there is something missing. The visual is only for you to visualize the link in RViz or Gazebo. Gazebo will need **<collision>** tags.

If you don’t have any collision property for your robot, then the robot will fall through the ground and continue falling indefinitely.

#### How to define a collision element

A <collision> tag will contain more or less the same thing as a <visual> tag: <geometry> and <origin> tags. You will basically define a shape.

As a general rule, for a collision, you will use a simpler shape than for the visual (if possible). The reason is simple—the more complex the shape, the more computation power will be required to compute the collision between the link and other elements. This could slow down the simulation a lot. Thus, designing simpler shapes is a best practice.

Here are a few more details about defining a collision element:

* If you are using complex Collada files (a few MB) for visuals, use simpler Collada or even STL files for the collisions. You can add those files to the meshes folder and include them in the URDF.
* If the shape is close to a basic one (a box, cylinder, or sphere), then you can use a basic shape for the collision. For example, if you design a mobile robot and the base of the robot looks like a box, then you can use a complex Collada or STL file for the visual, only using a box for the collision. For a wheel, you can use a cylinder, sphere, and so on.
* Even when using basic shapes, you could reduce the complexity—for example, by using a box for a collision when the visual is a cylinder or a sphere

To find real examples of this shape simplification, you can simply search on GitHub for existing projects.

Eg. google "github TurtleBot 3"

#### Adding collision properties for the links

You will add <collision> tags inside the <link> tags, in the mobile\_base.xacro file.

Here is the <collision> tag for base\_link:

<collision>

<geometry>

<box size="${base\_length} ${base\_width} ${base\_height}" />

</geometry>

<origin xyz="0 0 ${base\_height / 2.0}" rpy="0 0 0" />

</collision>

For wheels that you control and that touch the floor, it’s best to use a sphere for the collision, as this reduces the number of contact points with the ground (basically, just one contact point) and, thus, reduces the unwanted friction.

Let’s add a sphere collision for our wheels, inside the wheel\_link macro:

<collision>

<geometry>

<sphere radius="${wheel\_radius}" />

</geometry>

<origin xyz="0 0 0" rpy="0 0 0" />

</collision>

Finally, we add the collision for the caster wheel.

<collision>

<geometry>

<sphere radius="${wheel\_radius / 2.0}" />

</geometry>

<origin xyz="0 0 0" rpy="0 0 0" />

</collision>

#### Validating collision with Rviz

Open the **RobotModel** menu and uncheck the **Visual Enabled** box. Then, check the **Collision Enabled** box.

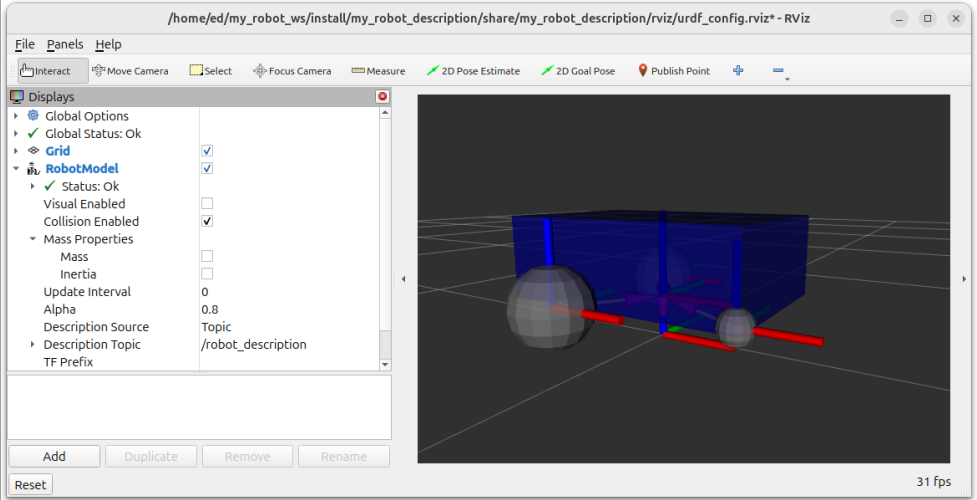


Figure 13.4: Visualizing collision in Rviz

There, you can see whether the collision properties are correct. In the **Links** menu (inside **RobotModel**), you can enable only some links if necessary, giving you a more precise view.

If you see that some collision elements are not correctly placed, or if they are too big or too small, you can then go back to your URDF and fix them.

As you can see in the preceding figure, the collision view for the robot is almost the same as the visual. The difference is with the wheels, which are now spheres.

## 13.4 Spawning the robot in Gazebo

(spawning refers to the act of creating and placing a new robot or object within the simulated environment)

(spawning = generate & deploy)

### Spawning the robot from the terminal

The first thing to start is the robot\_state\_publisher node:

$ ros2 run robot\_state\_publisher robot\_state\_publisher --ros-args -p robot\_description:="$(xacro /home/<user>/my\_robot\_ws/src/my\_robot\_description/urdf/my\_robot.urdf.xacro)"

After executing this command, the robot\_state\_publisher node starts and does three things — subscribes to /joint\_states, publishes on /tf, and also publishes the URDF on /robot\_description. You can verify this with **rqt\_graph** if needed

In Terminal 2, we start Gazebo — run the following command:

$ ros2 launch ros\_gz\_sim gz\_sim.launch.py gz\_args:="empty.sdf -r"

With this, we start an empty world in Gazebo.

We add the -r option to start the time directly so that we don’t have to click the play button.

Finally, we can spawn the robot in Gazebo. For this, we will use the **create** executable from the **ros\_gz\_sim** package.

In Terminal 3, run the following command:

$ ros2 run ros\_gz\_sim create -topic robot\_description

After running this command, you should see the robot in Gazebo:

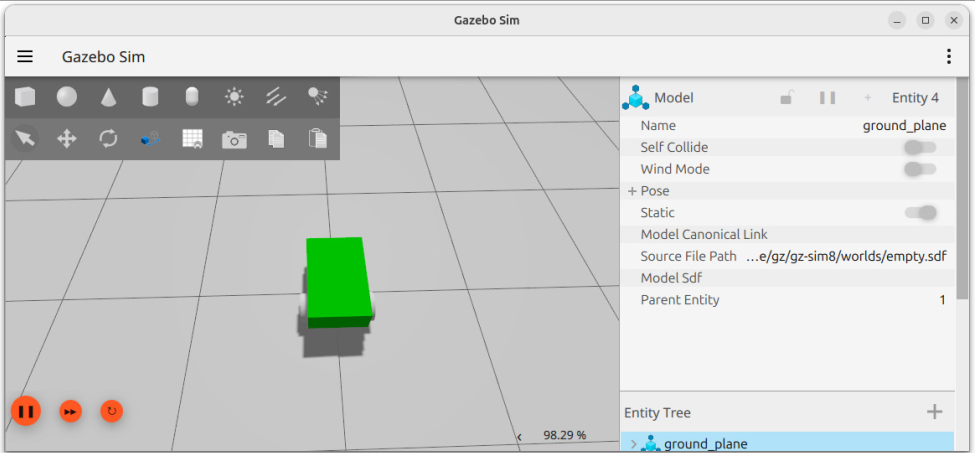


Figure 13.5: The robot spawned in Gazebo

If you get any error in any terminal, it probably means that your URDF is not correct.

### Spawning the robot from a launch file

We are now going to write a launch file to start those three commands.

#### Creating a \_bringup package

the best practice is to create a dedicated package for launch files and configuration files.

Following the naming convention for such a package, we will start with the name of the robot or application, adding the \_bringup suffix. Thus, we will create the my\_robot\_bringup package.

Let’s create this package, remove the unnecessary folders, and add a launch folder. We will also add a config folder, which we will use later in the chapter:

$ cd ~/my\_robot\_ws/src/

$ ros2 pkg create my\_robot\_bringup --build-type ament\_cmake

$ cd my\_robot\_bringup/

$ rm -r include/ src/

$ mkdir launch config

Now, in the CMakeLists.txt file of the my\_robot\_bringup package, add the instruction to install the launch and config folders:

install(

DIRECTORY launch config

DESTINATION share/${PROJECT\_NAME}/

)

The package is correctly set up, so we can now add and install files.

#### Writing the launch file (for gazebo)

Let’s create, write, and install the launch file to spawn the robot in Gazebo.

First, create a new file inside the launch folder. As this launch file will be the main one, let’s simply use the name of the robot (or robotics application), my\_robot.launch.xml:

$ cd ~/my\_robot\_ws/src/my\_robot\_bringup/launch/

$ touch my\_robot.launch.xml

Open the file and write the minimal code for an XML launch file:

<launch> </launch>

The beginning of the launch file will be very similar to the display.launch.xml file that we wrote in the previous chapter,

<let name="urdf\_path"

value="$(find-pkg-share my\_robot\_description)/urdf/my\_robot.urdf.xacro" />

Now, we can start the robot\_state\_publisher node:

<node pkg="robot\_state\_publisher" exec="robot\_state\_publisher">

<param name="robot\_description" value="$(command 'xacro $(var urdf\_path)')" />

</node>

Then, we start Gazebo with an empty world, and we also use the **-r** option to start the time automatically:

<include

file="$(find-pkg-share ros\_gz\_sim)/launch/gz\_sim.launch.py">

<arg name="gz\_args" value="empty.sdf -r" />

</include>

Finally, we spawn the robot in Gazebo:

<node pkg="ros\_gz\_sim" exec="create" args="-topic robot\_description" />

Now, save all the files, build the workspace, source the environment, and start the launch file (make sure that Gazebo is not running in another terminal before you do this):

$ ros2 launch my\_robot\_bringup my\_robot.launch.xml

The full my\_robot.launch.xml file would contain:

<launch>

<let name="urdf\_path"

value="$(find-pkg-share my\_robot\_description)/urdf/my\_robot.urdf.xacro" />

<let name="gazebo\_config\_path"

value="$(find-pkg-share my\_robot\_bringup)/config/gazebo\_bridge.yaml" />

<let name="rviz\_config\_path"

value="$(find-pkg-share my\_robot\_description)/rviz/urdf\_config.rviz" />

<node pkg="robot\_state\_publisher" exec="robot\_state\_publisher">

<param name="robot\_description"

value="$(command 'xacro $(var urdf\_path)')" />

</node>

<include file="$(find-pkg-share ros\_gz\_sim)/launch/gz\_sim.launch.py">

<arg name="gz\_args" value="empty.sdf -r" />

</include>

<node pkg="ros\_gz\_sim" exec="create" args="-topic robot\_description" />

<node pkg="ros\_gz\_bridge" exec="parameter\_bridge">

<param name="config\_file"

value="$(var gazebo\_config\_path)" />

</node>

<node pkg="rviz2" exec="rviz2" args="-d $(var rviz\_config\_path)" />

</launch>

Now, save all the files, build the workspace, source the environment, and start the launch file (make sure that Gazebo is not running in another terminal before you do this):

$ ros2 launch my\_robot\_bringup my\_robot.launch.xml

You should get the same result as when we ran all three commands in the terminal.

## 13.5 Controlling the robot in Gazebo

Our mobile robot is now simulated in Gazebo with physics properties. Now what? The robot is not doing anything. We will finish this chapter by adding control plugins so that we can simulate the hardware of the robot and do the following:

* Send commands to make the robot move in Gazebo, just as if it were in the real world
* Read all necessary joint states from the robot to get all the TFs in our ROS 2 application

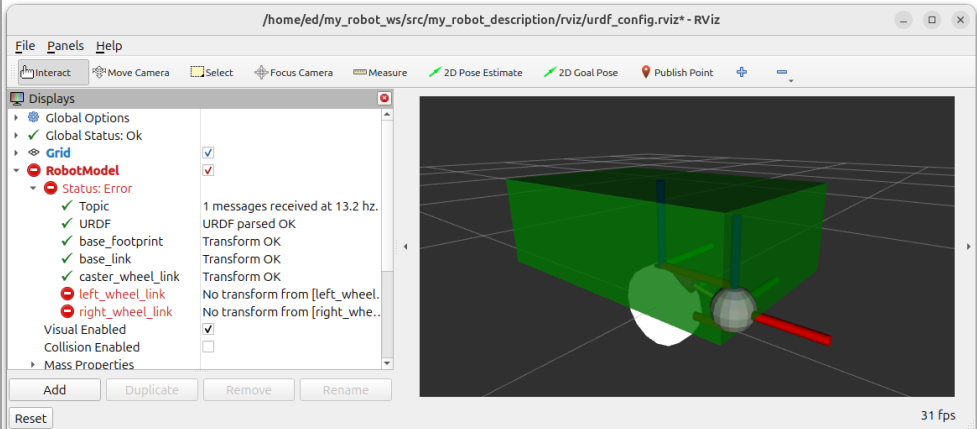
### What do we need to do?

When you start the my\_robot.launch.xml launch file, you see the robot in Gazebo. However, we don’t have any way to control it. In a terminal, if you list all nodes, topics, services, or even actions, you won’t find anything we can use.

to make things simple, you can start RViz using the previous configuration we saved:

$ ros2 run rviz2 rviz2 -d ~/my\_robot\_ws/src/my\_robot\_description/rviz/urdf\_config.rviz

You should see some errors in **RobotModel**, saying **No transform from [left\_wheel\_link]** and **No transform from [right\_wheel\_link]**.



*Figure 13.6: TF errors in RViz after spawning the robot in Gazebo*

This lack of TF is because nobody is publishing on the /joint\_states topic.

For a real robot, you would create a hardware driver to be able to control the wheels. This driver would expose a topic/service/action interface so that you could make the robot move. Then, you would read position/velocity data from encoders and publish this data on the /joint\_states topic.

For a Gazebo simulation, we will do the same thing but, of course, without hardware. We will use Gazebo plugins (also called systems) to simulate the control of the robot and get the joint states. Then, we will configure a *bridge* to make those plugins communicate with our ROS 2 application.

### Adding Gazebo systems

A Gazebo **system** is basically the simulation of a hardware component. You could have a system simulating a camera and publishing images, another one monitoring a battery state, and so on. For this book, we will use two systems—one to control a differential drive robot (two parallel wheels), and one to publish the joint states.

there are a lot of existing systems already available to use, including the two that we need.

For Gazebo Harmonic and ROS 2 Jazzy, you can find all available Gazebo systems here on GitHub:

<https://github.com/gazebosim/gz-sim/tree/gz-sim8/src/systems>

**Note:**

If there was not enough confusion already, on the internet you will often see the term ***plugin*** or ***system***; they both refer to the same thing. Even if the word system should be preferred, in practice it’s not clear which one to use; for example, to include a system in our code, we will need to use a <plugin> tag. So, in this section, I will have to use both terms.

#### Xacro file for Gazebo

The Gazebo systems for our robot will be specified in the URDF. So, we need to return to the my\_robot\_description package.

Our URDF is now split into three files: one with common properties, one with the description of the robot (links and joints), and one to include the other two.

To add the Gazebo systems, we will create yet another Xacro file, dedicated to all Gazebo-related stuff. By separating this file from the other ones, we make things cleaner

In the urdf folder of your my\_robot\_description package, add a fourth file, named mobile\_base\_gazebo.xacro.

Open the file and add the minimal Xacro code:

<?xml version="1.0"?>

<robot xmlns:xacro="http://www.ros.org/wiki/xacro">

</robot>

Now, in my\_robot.urdf.xacro, include the file after the two other ones:

<xacro:include filename="$(find my\_robot\_description)/urdf/mobile\_base\_gazebo.xacro" />

The Xacro file is ready, and we can now add the systems.

#### Differential drive controller

The first system we will add is a **differential drive controller**. By differential drive, we mean a robot controlled by two wheels, one on each side of the robot.

If you browse the available systems (the link is provided on the preceding page), you can find a diff\_drive folder—usually in ROS, we use diff drive as an abbreviation of differential drive.

In this folder, you will see a DiffDrive.hh file. Open this file, and there, near the beginning, you will find the XML tags related to the system.

Here is how to add the system to our Xacro file (mobile\_base\_gazebo.xacro):

<gazebo>

<plugin

filename="gz-sim-diff-drive-system"

name="gz::sim::systems::DiffDrive">

<left\_joint>base\_left\_wheel\_joint</left\_joint>

<right\_joint>base\_right\_wheel\_joint</right\_joint>

<frame\_id>odom</frame\_id>

<child\_frame\_id>base\_footprint</child\_frame\_id>

<wheel\_separation>0.45</wheel\_separation>

<wheel\_radius>0.1</wheel\_radius>

</plugin>

</gazebo>

We start with a <gazebo> tag. Everything related to Gazebo will be in such tags. Then, we include the system with a <plugin> tag.

Here is a bit more information about the different parameters for this diff drive system:

* left\_joint and right\_joint: You need to provide the exact name of the joints you have defined for the wheels in the URDF.
* **frame\_id**: As the robot moves, we will keep track of where it is relative to its starting position. This starting position will be called **odom** (short for **odometry**).
* child\_frame\_id: We write base\_footprint, as it is the root link for our robot and the one we want to use for odometry tracking.
* wheel\_separation: We can compute that from the URDF. The base width is 0.4, and the origin for each wheel is centered on the wheel. As each wheel length is 0.05, we need to add 0.4 + 0.025 + 0.025, which makes 0.45.
* wheel\_radius: We get this value from the URDF, which is defined as 0.1.

As the wheels turn and the robot moves, there will be some friction between the ground and the caster wheel.

So, we will reduce the friction for the caster wheel. You can add this code just before the code for the diff drive system:

<gazebo reference="caster\_wheel\_link">

<mu1 value="0.1" />

<mu2 value="0.1" />

</gazebo>

There are two parameters, mu1 and mu2, that you can set to have more control over the friction.

#### Joint state publisher

We have added a system to control the wheels, but before we test it, let’s finish the Xacro file and add the second system we need. The diff drive system alone won’t publish the joint states for the wheels; we need to add a joint state publisher system.

Let’s add the system to the Xacro file, after the previous one:

<gazebo>

<plugin

filename="gz-sim-joint-state-publisher-system"

name="gz::sim::systems::JointStatePublisher">

</plugin>

</gazebo>

### Bridging Gazebo and ROS 2 communications

#### What topics do we need to bridge?

from the ROS 2 side, we can’t communicate with Gazebo. We will need to create a bridge between ROS 2 and Gazebo using the ros\_gz\_bridge package.

To do that, we will run the parameter\_bridge node from the ros\_gz\_bridge package, with a configuration for the interfaces that we want to bridge.

#### Adding a configuration file to bridge topics

In the my\_robot\_bringup package, inside the config folder (that we already created before), create a new file named gazebo\_bridge.yaml.

Open this file to write the configuration. Here is the first bridge we will create:

- ros\_topic\_name: "/cmd\_vel"

gz\_topic\_name: "/model/my\_robot/cmd\_vel"

ros\_type\_name: "geometry\_msgs/msg/Twist"

gz\_type\_name: "gz.msgs.Twist"

direction: ROS\_TO\_GZ

Here are the different fields that we will use:

* ros\_topic\_name: The topic name on the ROS 2 side. Either you choose the topic name (/cmd\_vel doesn’t exist yet, so we create it) or you make it match with an existing one (for the next one, we will have to specify exactly /joint\_states).
* gz\_topic\_name: The topic name on the Gazebo size. We found it with gz topic -l.
* ros\_type\_name: The topic interface for ROS 2.
* gz\_type\_name: The topic interface for Gazebo. You can find it with gz topic -i -t <topic>.
* direction: Either ROS\_TO\_GZ, GZ\_TO\_ROS, or BIDIRECTIONAL. For example, /cmd\_vel is a topic that we publish in ROS 2 and subscribe in Gazebo, so we use ROS\_TO\_GZ. For /joint\_states, we publish in Gazebo and subscribe in ROS 2, so that will be GZ\_TO\_ROS. You can use BIDIRECTIONAL if you want to have publishers and subscribers on both sides of the same topic.

With this first bridge (above), we are able to send commands to the robot to make it move with the diff drive system.

Let’s now add the configuration for the /joint\_states topic (published by the joint state publisher system):

- ros\_topic\_name: "/joint\_states"

gz\_topic\_name: "/world/empty/model/my\_robot/joint\_state"

ros\_type\_name: "sensor\_msgs/msg/JointState"

gz\_type\_name: "gz.msgs.Model"

direction: GZ\_TO\_ROS

That will allow us to get all joint states for the robot and, thus, see the wheel TFs in RViz.

Finally, to get the odom to base\_footprint TF (published by the diff drive system), we also add this bridge:

- ros\_topic\_name: "/tf"

gz\_topic\_name: "/model/my\_robot/tf"

ros\_type\_name: "tf2\_msgs/msg/TFMessage"

gz\_type\_name: "gz.msgs.Pose\_V"

direction: GZ\_TO\_ROS

#### Starting the Gazebo bridge with the configuration

We can now add a new node to our my\_robot.launch.xml file to start the bridge, using the YAML configuration file we’ve just created.

First, at the beginning of the file, let’s add a new variable to find the path for the configuration file:

<let name="gazebo\_config\_path" value="$(find-pkg-share my\_robot\_bringup)/config/gazebo\_bridge.yaml" />

Then, after you spawn the robot in Gazebo with the create executable from ros\_gz\_sim, start the Gazebo bridge. You will need to pass the configuration file inside a config\_file parameter:

<node pkg="ros\_gz\_bridge" exec="parameter\_bridge">

<param name="config\_file"

value="$(var gazebo\_config\_path)" />

</node>

As we use the ros\_gz\_bridge package inside my\_robot\_bringup, we will also add a new dependency inside the package.xml file:

<exec\_depend>ros\_gz\_bridge</exec\_depend>

### Testing the robot

Send a velocity command from the terminal:

$ ros2 topic pub /cmd\_vel geometry\_msgs/msg/Twist "{linear: {x: 0.5}}"

The robot should start moving in Gazebo (to stop, send the same command with {x: 0.0}). If you see the robot moving, it means that the bridge is correctly configured, as the ROS 2 topic can reach the Gazebo system. It also means that the diff drive system works.

To achieve a better way to control the robot and make more tests, you can run this node instead:

$ ros2 run teleop\_twist\_keyboard teleop\_twist\_keyboard

This will listen to your keyboard and publish to the /cmd\_vel topic (if you use a different name for the topic, simply add a remapping with --ros-args -r).

## 13.6 Summary

# Chapter 14 - Going Further – What To Do Next

## 14.1 ROS 2 roadmap – exploration phase

### Common stacks and frameworks

After learning the core programming basics and concepts, such as TF and URDF, a very common next step is to learn about some of the existing ROS 2 **stacks** and **frameworks**, and also learn how to create interfaces between ROS 2 and hardware components.

**Note:**

You will often see the terms *stack*, *framework*, and other variations. They usually mean the same thing. Basically, they are collections of packages that focus on solving a specific problem.

Among those stacks/frameworks, you can find ***Navigation 2*** (for mobile robots), ***MoveIt 2*** (for robotic arms and grippers), and ***ros2\_control*** (for hardware control).

#### Hardware interface (and ros2\_control)

#### Navigation 2 stack

The **Navigation 2** stack, also known as **Nav2**, is very popular for a reason: most robots using ROS 2 are mobile robots

Fortunately, you can use the Nav2 stack. With this stack, you can easily create a map of the environment with **Simultaneous Localization And Mapping (SLAM)**, and then use this map to make your robot navigate from one place to another, while avoiding obstacles.

#### MoveIt 2

a stack for robotic arms.

MoveIt 2 will do the motion planning for you, for a robotic arm, or even a system with several robotic arms. It also has functionalities for grasping.

### More exploration topics

#### More advanced ROS 2 concepts

* **Lifecycle nodes** (also called **managed nodes**): These nodes contain a state machine that allows you to easily separate your code for different parts of initialization and activation. This is especially useful when dealing with hardware. For example, you can make sure that a hardware component is correctly connected and initialized before using it in a critical part of your application. Also, lifecycle nodes will be useful if you want to learn ros2\_control.
* **Executors**: With executors, you can have more control over how callbacks are handled within a node or several nodes
* **Components**: By making your nodes components, you can run several nodes from within one executable. This can reduce resource usage and speed up communication. To learn about components, you first need to understand executors. Then, components will also help you understand ros2\_control.

## 14.2 Learning for a specific goal

### What to learn for a project?

### What to learn to get a job?

# Summary

If you liked this book and the way I teach, here are a few more resources from me:

* Robotics Backend website (<https://roboticsbackend.com>): Here, you will find more written tutorials about ROS 2 and other robotics-related topics
* Robotics Backend YouTube channel (<https://www.youtube.com/c/RoboticsBackend>): For video tutorials and free crash courses
* Full-length online courses (<https://roboticsbackend.com/courses>): I also provide complete ROS 2 courses that you can purchase, with a strong focus on practical learning

# Other Books You May Enjoy