



# Welcome to ROS2 learning resources!

## 1. Basics of ROS

### First, What and Why ROS?

As defined by its developers, 'The Robot Operating System (ROS) is a set of software libraries and tools that help you build robot applications. From drivers to state-of-the-art algorithms, and with powerful developer tools, ROS has what you need for your next robotics project. And it's all open source.'

(Although ROS stands for Robot Operating System, it is not actually an OS! its a framework)

The development of ROS is overseen by [Open Robotics](#) You can learn more about ROS on its official [website](#)

There is another video which inspire you to move to ROS

### What is ROS? Why it's Important for making Robots!

In recent years the ROS project has undergone a significant upgrade, and newer releases come under the name "ROS 2". A significant portion of the documentation and tutorials available online is for the original ROS (now sometimes called ROS 1), however ROS 1 will no longer be receiving updates. And hence, in our theme we will be working with ROS2 Humble, which is one of the many [ROS2 distributions](#) released every year!



With no further ado, let's get started with ROS2.

This page contains a broad overview of some key concepts that you will be using in almost every robotics project:

1. Nodes
2. Topics and Messages
3. Services
4. Launch Files
5. Packages
6. Workspace

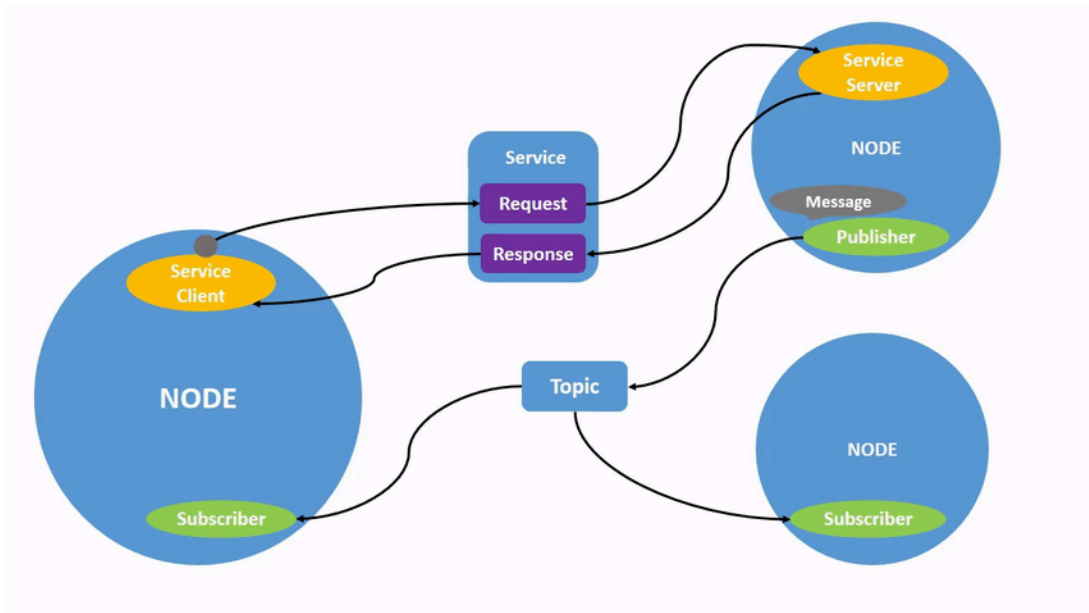
# ROS Concepts

## 1. Nodes

A ROS system is made up of a bunch of smaller programs that all run at once and talk to each other. Each of these programs is called a node. A ROS node is very similar to any other command-line program, except that it is able to utilise the ROS libraries, and other parts of the ROS ecosystem will be aware of it. A node is typically designed to perform one specific task as part of the larger system. Some example tasks a node could perform are:

- Reading data from a sensor
- Sending control signals to a motor
- Computing a trajectory to follow
- Receiving operator controls from a joystick
- Displaying a visualisation to an operator

Each ROS node on a computer, and on other computers on the same network, all form part of a ROS network where they can “see” each other. This gives us a huge level of flexibility as we can just as easily develop systems that run entirely on a single machine, or with components distributed between different machines.



In the above illustration, you can see that a full robotic system is comprised of many nodes working in concert. In ROS 2, a single executable (C++ program, Python program, etc.) can contain one or more nodes.

Two ROS commands to work with nodes:

- `ros2 run <package_name> <node_name>` - To run a node (we'll talk more about the package name later)
- `ros2 node list` - To see all the nodes currently running on our network.

In the example below, we run a node named `turtlesim_node` from the package **turtlesim** (which is the most common tool used to get familiarized with ROS) using `ros2 run` command. After which if we do `ros2 node list`, we will see a `/turtlesim` node. This command shows that the `/turtlesim` node is running. Now try it yourself! Run the following commands in separated terminals:

```
ros2 run turtlesim turtlesim_node
```



```
ros2 node list
```



What was not mentioned here was when you run the `ros2 run turtlesim turtlesim_node` command, a blue window opens up with a random image of a turtle in its center. That is our turtle and its name is **turtle1**. In further topics, you will learn how you can move, spawn, reset, kill (even if you don't want to 🤨) your turtle and many more!

### Fun experiment 🤩 !!!

Now let's try a fun experiment! Keep your node running and connect one of your team members' computer with the same Wi-Fi network and try running `ros2 run turtlesim turtle_teleop_key` without running **turtlesim** package in their computer. Now, press the arrow keys. And voila! you just the turtle in **/turtlesim** node even though its on a different computer! That's how powerful ROS is!

```
erts@erts: ~
erts@erts:~$ ros2 run turtlesim turtlesim_node
Warning: Ignoring XDG_SESSION_TYPE=wayland on Gnome. Use QT_QPA_PLATFORM=wayland
to run on Wayland anyway.
[INFO] [1694591933.537596672] [turtlesim]: Starting turtlesim with node name /tu
rtlesim
[INFO] [1694591933.548908681] [turtlesim]: Spawning turtle [turtle1] at x=[5.544
445], y=[5.544445], theta=[0.000000]
```

```
erts@erts: ~
erts@erts:~$ ros2 node list
/turtlesim
erts@erts:~$
```

In figure given below, you can see **/turtlesim** and **/teleop\_turtle** nodes recognized in a single terminal despite being on different computers!

```
ertslab@ertslab-ThinkCentre-E73z:~$ ros2 node list
/teleop_turtle
/turtlesim
ertslab@ertslab-ThinkCentre-E73z:~$
```

## 2. Topics and Messages

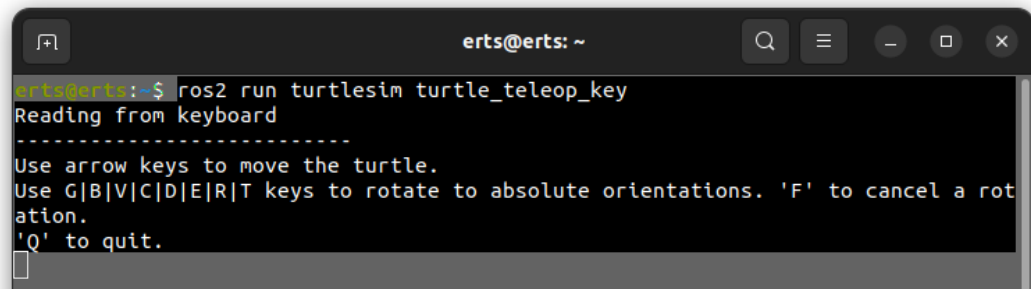
The way ROS nodes communicate with each other is with topics and messages. A topic is a named location that one (or occasionally multiple) nodes can publish a message to. These nodes are called publishers, and other nodes (called subscribers) can subscribe to the topic to receive them. A node may be a publisher for one topic, and a subscriber for a different topic.

The topic system is great because it lets us write smaller, simpler nodes that are good at one thing, and can communicate with each other to solve larger problems. Let us now extend our example a little further, run the following command in a new terminal to create another node that is **/teleop\_turtle** (you can verify the node is running in a separate terminal tab/window by using `ros2 node list`):

```
ros2 run turtlesim turtle_teleop_key
```



You will get the following output:



```

erts@erts: ~
erts@erts:~$ 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.

```

Now while keeping this terminal active, try pressing the arrow keys on your keyboard and notice the turtle. You will see that the turtle starts to move like an RC car!

**NOTE: the key press must be made with the cursor focused on the teleop window. (Not on the turtlesim window)**

The **/teleop\_turtle** node only needs to know how to talk to the **/turtlesim** node and publish a **Twist** message (Twist message is used to send linear and angular values to move any body). It doesn't care what anyone else does with the message afterwards. Likewise, the **/turtlesim** node in which the turtle moves doesn't need to know how to communicate with every node out there, it just needs to subscribe to a topic with **Twist** messages.

The **/teleop\_turtle** node only needs to know the msg type to publish to the **/turtle1/cmd\_vel** topic. i.e. It will publish a **Twist message** (Twist message is used to send linear and angular values to move any body) to **/turtle1/cmd\_vel** topic. It doesn't care what anyone else does with the message afterwards. Likewise, the **/turtlesim** node in which the turtle moves doesn't need to know how to communicate with every node out there, it just needs to subscribe to the **/turtle1/cmd\_vel** topic with **Twist** messages.

Few useful commands for working with topics are:

- `ros2 topic list` - See all the available topics being published to
- `ros2 topic echo <topic name>` - Pretend we are a subscriber and see the messages being published.
- `ros2 topic info <topic name>` - Gives the message type with publisher and subscriber count.

In the below example, you can see **/teleop\_turtle** node we ran earlier is publishing to the topic **/turtle1/cmd\_vel** when we press arrow keys.

```

erts@erts: ~
erts@erts:~$ ros2 topic list
/parameter_events
/rosout
/turtle1/cmd_vel
/turtle1/color_sensor
/turtle1/pose
erts@erts:~$ ros2 topic echo /turtle1/cmd_vel
linear:
  x: 2.0
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 0.0
---
linear:
  x: 0.0
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 2.0

```

```

erts@erts: ~
erts@erts:~$ ros2 topic info /turtle1/cmd_vel
Type: geometry_msgs/msg/Twist
Publisher count: 0
Subscription count: 1

```

**NOTE:** Learn more about topics, publishers and subscribers [here](#).

### 3. Services

Unlike topics, which allow a node to share a constant stream of messages to anyone who cares to listen, services offer a single request/reply communication from one node to another. The content of the reply may be useful information, such as the result of a calculation, or it may just be an indication that the request has been received.

To use our turtle as an example, let's say we want to spawn another turtle. This can be achieved by a single message to spawn a turtle and a constant communication like that of a publisher and subscriber is not needed. For such cases we do not use ROS topics. We just call a service for our **/turtlesim** node to spawn another turtle. You can do that by running the following command:

```
ros2 service call /spawn turtlesim/srv/Spawn '{"x': 2.0, 'y': 2.0, 'theta': 0.0}"
```

You will notice that another turtle has spawned near the lower left corner corner of your screen! And when you run `ros2 topic list`, you will see that new topics have been created for our new **turtle2**.

Again, two useful commands for working with services are:

- `ros2 service list` - To see a list of all services available
- `ros2 service call <service_name> <service_message>` - To manually call a service with the command.

In our above example, we call a service called **/spawn** to spawn the new turtle and this service sends the **turtlesim/srv/Spawn** message type which contains all the info. required to spawn the new

turtle. You can try various other services on your own. Say, try the `/reset` service and it shall reset your **turtle1** to its initial position and orientation.

```

erts@erts: ~
erts@erts:~$ ros2 service list
/clear
/kill
/reset
/spawn
/teleop_turtle/describe_parameters
/teleop_turtle/get_parameter_types
/teleop_turtle/get_parameters
/teleop_turtle/list_parameters
/teleop_turtle/set_parameters
/teleop_turtle/set_parameters_atomically
/turtle1/set_pen
/turtle1/teleport_absolute
/turtle1/teleport_relative
/turtlesim/describe_parameters
/turtlesim/get_parameter_types
/turtlesim/get_parameters
/turtlesim/list_parameters
/turtlesim/set_parameters
/turtlesim/set_parameters_atomically
erts@erts:~$ ros2 service call /spawn turtlesim/srv/Spawn '{"x": 2.0 , 'y': 2.0 ,
'theta': 0.0}"
waiting for service to become available...
requester: making request: turtlesim.srv.Spawn_Request(x=2.0, y=2.0, theta=0.0,
name='')

response:
turtlesim.srv.Spawn_Response(name='turtle5')
```

#### 4. Launch Files

When we're working on a ROS project we will often be running the same nodes over and over again with the same parameters and remappings. This becomes pretty tedious as you need to remember to open all the terminals and get all the parameters right, then stop them all when you're done.

To help with this, ROS provides a scripting system called "launching" 🚀 that lets us configure and launch a bunch of nodes together in a group. In ROS 1 the launch system was based on XML files, and although ROS 2 does technically support the XML format, currently the more popular and powerful approach is a Python-based system.

Unfortunately, the syntax of the Python scripts can also be pretty confusing to wrap your head around. And for that reason, we won't be going into much further details for now.

For now, just know that the two ways to run a launch file are:

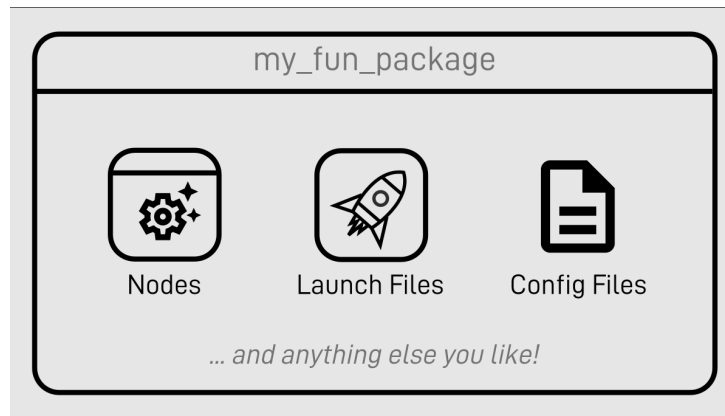
- `ros2 launch <package_name> <launch_file_name>` - To run a launch file from a package
- `ros2 launch /path/to/launch/file` - To run a launch file directly (useful for a quick test, but putting into a package is better long-term).

#### 5. Packages

Ok, so we know that a ROS system is made up of all these nodes and parameters and launch files and so on, but how are these organised?

The first level of organisation is called a package. A package is a way that we can group a bunch of closely related files together, and especially (but not always) so that we can reuse common code in

different projects. A package can contain a variety of things, such as a single node, a set of nodes and configuration files, models for a robot, software libraries, or anything else you want!



In addition to its own files, each package includes a list of which other packages it depends on, things it needs in order to work which are known as **dependencies**.

Let us recall the syntax we shared in the **Nodes** section:

```
ros2 run <package_name> <node_name>
```



In our example, the name of package we are using is **turtlesim**. And we run the **/turtlesim\_node** which is present inside the **turtlesim** package.

```
ros2 run turtlesim turtlesim_node
```



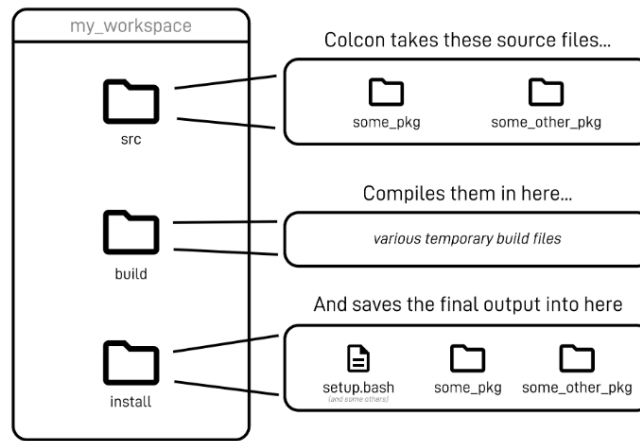
## 6. Workspace



When we work on ROS projects we create something called a workspace to keep all our own packages inside it. You can manage this however you want, but it generally makes sense for each project to have its own workspace, which will contain whatever packages are relevant to that project. Note that here we're talking about packages to be compiled from source (either written ourselves or pulled from somewhere like GitHub), the system-installed packages will be visible to all our workspaces.

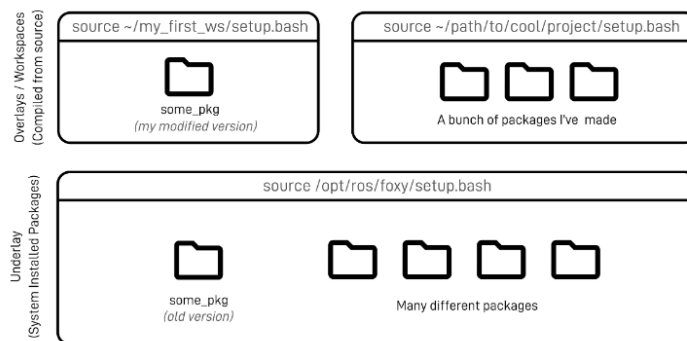
If we were to build our code using normal tools, they wouldn't necessarily know where to find all the ROS components, and more importantly ROS wouldn't know where to find our packages. This is where the build tool `colcon` comes in.

When we use `colcon` to build our workspace, it will take all our source files from a `src` directory, build them into a build directory, and "install" them into an install directory. This isn't too different from other build tools, however it also creates a `setup.bash` file with all the information the system needs to how to find our packages, just like we had for the system installation. So every time we open a terminal to run code from that project, we need to source that workspace by running `source path/to/workspace/setup.bash`.



We sometimes call this the overlay, because it goes over the underlay. With the overlay sourced, ROS will first look for packages in our workspace (the overlay), before looking at the system installation (the underlay). This way we can test against modified/updated versions of base packages without having to uninstall them. We don't usually want to automatically source this using our `~/ .bashrc` file, since we want to keep each workspace separate.

There's nothing stopping you from stacking overlays on top of each other, with each treating the previous as an underlay, but you'd only need that for complex projects.



When we build using colcon, it doesn't matter whether our packages are written in C++ or Python, and if the dependencies are simple or complex, it just figures everything out for us. This whole process may seem a bit complicated at first, but it is easy to get used to, and once we have projects with more than a couple of packages it makes things much simpler.

That is all for our summary of very important concepts from ROS. But we would strongly recommend going through the ROS tutorials from the official docs to get a much better understanding of ROS. Specifically the beginner section, link below:

[ROS2 Beginner Guide.](#)

## 2.URDF :

When we create a robotic system, there may be many different software components that need to know about the physical characteristics of the robot. So in ROS URDF files are used. **URDF** (Unified Robot Description Format) is a file format for specifying the geometry and organization of robots in ROS.

For more details look [here](#).

Also you can refer this [website](#).



### References and additional resources

- [Getting Ready for ROS Part 4: ROS Overview \(10 concepts you need to know\)](#)
  - [ROS2 Wiki \(Humble Documentation\)](#)
  - [What Is ROS2? - Framework Overview](#)
  - [ROS1 vs ROS2](#)
- 

All the best 👍 !!

