Introduction to ROS2::

ROS 2 is an open-source framework designed to support the development of robot software systems. It is the successor of ROS 1, redesigned to overcome limitations like lack of real-time support, multi-platform issues, and poor security. ROS 2 provides a modular, scalable, real-time capable middleware for building industrial-grade robot applications.

ROS 2 is built on top of DDS(Data Distribution Services), a standard for real time, scalable, and high-performance communication.
Key architectural components include: Nodes, Topics, Services, Actions, Parameter, and launch system.

What is a Node?
A Node is a process that performs computation. In ROS 2, multiple Nodes can be launched in a single process(called a composable node), enhancing performance.
“In simple terms a node is small program that runs and handles a specific task in a robotic system.”

Each Node is responsible for one function(e.g., reading sensor data, controlling motors, processing images).
Nodes communicate with each other using Topics, Services, and Actions.
Example: A talker node might send a message and a listener node might receive that message.

We can create the most simple talker and listener node and can see how they transmit and receive the message, for that, type command→
ros2 run demo_nodes_cpp talker (for talker node, in first terminal)
ros2 run demo_nodes_cpp listener (for listener node, in second terminal)

to see the publisher, subscriber and topic, you need to start rqt_graph, for that type command →
rqt_graph (for RQT graph, in third terminal)

These were some example of Nodes, also we can create Nodes using Python, oops. For example lets create a Node using python:

```
import rclpy
from rclpy.node import Node

class HelloNode(Node):
    def __init__(self):
        super().__init__('hello_node')
        self.get_logger().info("Hello ROS 2!")

rclpy.init()
node = HelloNode()
rclpy.spin(node)
node.destroy_node()
rclpy.shutdown()
```

This is a node, created using python, and it prints a message “HELLO ROS 2!” to the terminal using log.

What is a Topic?
A Topic is a communication channel which is used by nodes to send and receive message/data asynchronously.

A Topic is like a radio frequency—one node can publish data to it and other nodes can subscribe to receive the data.

Topics enable asynchronous, one-to-many communication between nodes. A node can publish the message to a topic and then another node which subscribe the topic can receive the message, there can be one or more subscribers which will be getting the data.
Publisher: sends messages
Subscriber: receives messages

Characteristics:
1) Used for one-to-many or many-to-many communication.
2) Communication is unidirectional means, data flow only from publisher to subscriber.
3) Messages are exchanges in real-time using predefined message types(e.g., sensor data, integers, strings, ___images).

For example:
A node temperature_sensor publishes data to a topic named /temp.
Another node display_node subscribes to /temp and shows the readings.

What are Services?
A service is a way for nodes to communicate using a request-response pattern.
A service is like asking a question and then waiting for the response, one node sends a request, and another node sends back a response.

Services are useful for two way communications, unlike Topics(which are one way), services are synchronous.
These are used when a node needs a specific task done immediately and waits for the result.

For example:
A node led_client sends a request to turn on an LED using the service /toggle_led.
The node led_server receives the request and replies with a status(e.g., LED ON).

Services are used when a command or query need confirmation or a result(e.g., start/stop motor).

What are Actions?
An action is used when a node needs to perform a task that might take time and should provide feedback during execution.
An action is like asking a robot to do something that take time, such as moving to a specific location from a point(e.g., from origin to a point (50,50))-- while doing the task, it can keep you updated on progress, and you can even cancel it.
For tasks which take a few seconds or longer, need a progress update, and might need to be cancelled, we should go for actions rather than services

These are built on top of services but designed for asynchronous and cancelable operations.
Example:
A node move_client sends a goal to an action server /move_to_point to move the robot to a point.

The server gives feedback like “10% completed”, and finally replies “Arrived”.
The client can cancel if needed.

Advantages of using ROS2 over a simple python script::

Feature	ROS 2	Simple Python Script
1. Modularity	Code is split into nodes, each with a specific job. Easier to debug, reuse, and scale.	Everything in one file or multiple tightly coupled scripts. Harder to manage as complexity grows.
2. Communication	Built-in support for **inter-process communication** via topics, services, actions.	Must manually manage communication (sockets, serial, etc.), which is error-prone.
3. Scalability	Designed for **multi-robot systems** and large distributed architectures.	Not scalable beyond simple single-machine or one-robot setups.
4. Real-Time Support	Integrates with DDS (Data Distribution Service) to support real-time communication.	Python scripts aren’t real-time by default; timing is managed manually.
5. Tools and Ecosystem	Comes with tools like `rviz2`, `ros2bag`, `rqt`, and `tf2` for visualization, logging, simulation, etc.	No built-in tools; must write everything from scratch or use third-party libraries.
6. Parameter Handling	Dynamically update node behavior using parameters at runtime.	Changing script behavior often requires restarting or editing code.
7. Standardization	Encourages clean architecture and common design patterns in robotics.	Every developer may structure code differently. Harder to maintain as a team.
8. Middleware Flexibility	Uses DDS for flexible and efficient communication (QoS, reliability, etc.)	You must implement your own communication logic (serial, sockets, etc.).
9. Interoperability	Nodes written in different languages (Python, C++, Rust) can work together.	Hard to integrate code in multiple languages without extra overhead.
10. Simulation & Integration	Easy integration with Gazebo, MoveIt, Navigation2, etc.	Manual simulation/integration, often from scratch.

Installation and setting up packages::

To install ROS2 jazzy on your UBUNTU 24.04 version follow these commands:

sudo apt update
sudo apt install software-properties-common
sudo add-apt-repository universe

sudo apt install ros-apt-source

ros-apt-source add --rosdistro jazzy

sudo apt update
sudo apt install ros-jazzy-desktop

#If you are facing any problem, you can go checkout to the official ROS2 website for installation, here is the link:
https://docs.ros.org/en/jazzy/Installation.html

After completing the installation to check the successful installation you can run these commands:

ros2 run demo_nodes_cpp talker # it is a node that will start publishing "Hello World "
ros2 run demo_nodes_cpp listener # a subscriber which will listen to talker node.

Once the installation is complete you need to make ros2 workspace, building and installing the packages required→

 # post installation setup **

1) Open the terminal and type → **sudo apt install python3-colcon-common-extensions**

2) type → nano .bashrc # this will open a file in the end of the file add the line →
source /opt/ros/jazzy/setup.bash #and then press ctrl + o then enter then ctrl + x.

3) Open the terminal and type→ **mkdir ros2_ws** # this will create a folder named ros2_ws, you can choose any name but it is best practice to choose this as your workspace folder.

4) Now type→ **cd ros2_ws** #you will get inside the ros2_ws folder.

5) type → **mkdir src** #creates a directory named src

6) type→ **cd src** #get inside of src folder.

7) type → **nano .bashrc** **#** again a file will be opened and add this line to the end of file
add → **source ~/ros2_ws/install/setup.bash** #and then press ctrl + o then enter then ctrl + x.

8) type→ **ros2 pkg create your_pkg_name –build-type ament_python –dependencies rclpy** (for python pkg)
or
type→ **ros2 pkg create your_pkg_name –build-type ament_cmake –dependencies rclcpp** (for cpp pkg)

9) type→ **cd your_pkg_name/your_pkg_name/** #get inside your packages

10) type→ **touch your_node_name** # you have created a node file, now just put your code

since you have created all the necessary packages and folders now for writing the code you will need a code editor for that open the terminal and type

1) **sudo apt install snap** #install latest version
2) **sudo snap install codium** #in some cases it will ask to install classic, then just add '—classic' ahead of it

To open the code editor just type "codium" in terminal.

Open the src folder from ros2_ws in you editor there you will see your packages that you built, now open your node file and write the code for your node.

 # Some more useful installs ::

Note :: It is advisable to run these commands before running a node if you have made some changes to the node code →

open terminal and go insider ros2_ws directory and
type → **colcon build**

after the process is finshed now

type → **source install/setup.bash**

If you are getting error while running the command **" colcon build"** , then you might need to install some more packages. For that open the terminal and type

→ **sudo apt install python3-pip**

and now you are all set to run your written node.

Writing simple nodes::

Write a simple node that will publish a message to a topic →

1) Create a package: Open a new terminal and source your ros2 installation so that your ros2 command will work.

Navigate to you ros2_ws directory and then again navigate to your src folder, keep that in mind that the package should be built inside the src folder.

Now run this command to create a python package:

Type → **ros2 pkg create pub_sub –build-type ament_python –dependencies rclpy**

2) To write the node navigate to **“ ros2_ws/src/pub_sub/pub_sub ”,** to navigate there open terminal and

type → **cd ros2_ws/src/pub_sub/pub_sub/**

Inside the package create a node file named **publisher.py**, to do that

type → **touch publisher.py**

To make it executable, type → **chmod +x publisher.py**

Now open your code editor, if you have installed codium then type “ **codium ”** in a new terminal.

Open the src folder in your codium and navigate to your node file (i.e., publisher.py).

Write this code to your file publisher.py

Note: This node is only a publisher node which will publish a specific message to a topic, the code for the subscriber node will be written in another file.

Python code for publisher node is written on the next page →

 # code for publisher node::

#**python**

```
import rclpy                                    # python ros2 client library

from rclpy.node import Node                     # to use the node class of rclpy

from std_msgs.msg import String                 # to use String of std_msgs.msg


class MinimalPublisher(Node):                   # declare the class which will inherit from the Node class

    def __init__(self):                         # constructor of MinimalPublisher class
        super().__init__('minimal_publisher')
        self.publisher_ = self.create_publisher(String, 'topic', 10)        # created a publisher which publish string data
        timer_period = 0.5  # seconds                                       # time period for timer
        self.timer = self.create_timer(timer_period, self.timer_callback)      # created timer which calls timer_callback
        self.i = 0                                                           # after each time period.

    def timer_callback(self):                                                # define timer_callback function
        msg = String()                                                       # msg type = string
        msg.data = 'Hello World: %d' % self.i                                # msg data = hello world 0
        self.publisher_.publish(msg)                                         # publish the message to topic
        self.get_logger().info('Publishing: "%s"' % msg.data)                # print published message to screen
        self.i += 1


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

    minimal_publisher = MinimalPublisher()                                   # class name

    rclpy.spin(minimal_publisher)    # spin the node i.e. do not kill after one operation

    # Destroy the node explicitly
    # (optional - otherwise it will be done automatically
    # when the garbage collector destroys the node object)
    minimal_publisher.destroy_node()
    rclpy.shutdown()


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

Note: After writing the node, save the file and go to **setup.py,** in the same folder and add this line in the section

"entry points" → **"publisher=pub_sub.publisher:main",** and now save the file.

Writing the subscriber node:

1) Navigate to **cd ros2_ws/src/pub_sub/pub_sub/**

2) Type → **touch subscriber.py**

3) Type → **chmod +x subscriber.py**

4) Type → **codium**

 # Now write this code inside the subscriber.py file ::

#python

```
import rclpy                                    #ROS2-python client library
from rclpy.node import Node                     # to use the node class of  rclpy

from std_msgs.msg import String                 # to use string datatype of std_msgs


class MinimalSubscriber(Node):                  # declared the class which will inherit from node class

    def __init__(self):                         # class constructor
        super().__init__('minimal_subscriber')  # Node class constructor
        self.subscription = self.create_subscription(   # created a subscriber which will subscribe every time when a
            String,                             # a message of string type is published to topic and calls the
            'topic',                            # listener_callback function
            self.listener_callback,
            10)
        self.subscription                       # prevent unused variable warning

    def listener_callback(self, msg):           # defined listener_callback function
        self.get_logger().info('I heard: "%s"' % msg.data)


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

    minimal_subscriber = MinimalSubscriber()

    rclpy.spin(minimal_subscriber)

    # Destroy the node explicitly
    # (optional - otherwise it will be done automatically
    # when the garbage collector destroys the node object)
    minimal_subscriber.destroy_node()
    rclpy.shutdown()


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

Note do the same thing as we did for publisher node, open **setup.py** and in the " entry points" block add this line

→ **"subscriber=pub_sub.subscriber:main",**