

ROS2 GUIDE

 Ubuntu 22.04, ROS2 Humble

 Andrea Ruo

Table of contents

1. [Installation](#)

1. [Make sure you have a locale which supports UTF-8](#)
2. [Add the ROS 2 apt repository to your system](#)
3. [Install ROS2 packages](#)
4. [Environment setup](#)
5. [Try Talker-Listener example](#)

2. [Concepts](#)

1. [Nodes](#)
2. [Discovery](#)
3. [Interfaces](#)
4. [Topics](#)
5. [Services](#)
6. [Actions](#)
7. [Parameters](#)
8. [Introspection with command line tools](#)
9. [Launch](#)
10. [Client Libraries](#)

3. [Command Line Interface Tools](#)

1. [Understanding Nodes](#)
2. [Understanding Topics](#)
3. [Understanding Services](#)
4. [Understanding Parameters](#)
5. [Understanding Actions](#)
6. [rqt_console](#)
7. [ros2 bag](#)
8. [Verification and installation packages](#)
9. [Clock](#)

4. [Workspace and Packages](#)

1. [Create a Workspace](#)
2. [Create a Package](#)
 1. [C++ package](#)
 2. [Python package](#)

3. [Costumize a Package](#)
 4. [Build Workspace](#)
 5. [Add personal / external library](#)
 6. [Source workspace](#)
-
5. [Create a Publisher and Subscriber](#)
 1. [C++ Publisher and Subscriber](#)
 2. [Python Publisher and Subscriber](#)
-
6. [Create my_interfaces package](#)
-
7. [Create a Publisher and Subscriber with my_interfaces](#)
 1. [C++ Publisher and Subscriber with my_interfaces](#)
 2. [Python Publisher and Subscriber with my_interfaces](#)
-
8. [Create Service and Client using my_interface](#)
 1. [C++ Service and Client with my_interfaces](#)
 2. [Python Service and Client with my_interfaces](#)
-
9. [Create Action Server and Client using my_interface](#)
 1. [C++ Action Server and Client with my_interfaces](#)
 2. [Python Action Server and Client with my_interfaces](#)

Installation (DEBIAN)

1. Make sure you have a locale which supports UTF-8

```
locale # check for UTF-8

sudo apt update && sudo apt install locales
sudo locale-gen en_US en_US.UTF-8
sudo update-locale LC_ALL=en_US.UTF-8 LANG=en_US.UTF-8
export LANG=en_US.UTF-8

locale # verify settings
```

2. Add the ROS 2 apt repository to your system

First ensure that the Ubuntu Universe repository is enabled.

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

Now add the ROS 2 GPG key with apt.

```
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
```

Then add the repository to your sources list.

```
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
```

3. Install ROS2 packages

Update your apt repository caches after setting up the repositories. It is always recommended that you ensure your system is up to date before installing new packages.

```
sudo apt update  
sudo apt upgrade
```

Desktop Install (Recommended): ROS, RViz, demos, tutorials.

```
sudo apt install ros-humble-desktop
```

4. Environment setup

Set up your environment by going to `.bashrc` and typing at the end:

```
# source ros2  
export ROS_DOMAIN_ID=0  
source /opt/ros/humble/setup.bash  
source /usr/share/colcon_argcomplete/hook/colcon-argcomplete.bash
```



The `.bashrc` file is located in the **Home** folder as hidden file.

5. Try Talker-Listener example

In one terminal run:

```
ros2 run demo_nodes_cpp talker
```

In another terminal run:

```
ros2 run demo_nodes_py listener
```

You should see the talker saying that it's Publishing messages and the listener saying I heard those messages. This verifies both the C++ and Python APIs are working properly.

Concepts

Nodes

A **node** is a participant in the ROS 2 graph, which uses a client library to communicate with other nodes. Nodes can communicate with other nodes within the same process, in a different process, or on a different machine. Nodes are typically the unit of computation in a ROS graph; each node should do one logical thing.

Nodes can:

- Publish to named **topics** to deliver data to other nodes;
- Subscribe to named **topics** to get data from other nodes;
- Act as a **service client** to have another node perform a computation on their behalf;
- Act as a **service server** to provide functionality to other nodes. For long-running computations;
- Act as an **action client** to have another node perform it on their behalf;
- Act as an **action server** to provide functionality to other nodes.

Connections between nodes are established through a distributed discovery process.

Discovery

Discovery of nodes happens automatically through the underlying middleware of ROS 2. It can be summarized as follows:

When a node is started, it advertises its presence to other nodes on the network with the same ROS domain (set with the **ROS_DOMAIN_ID** environment variable). Nodes respond to this advertisement with information about themselves so that the appropriate connections can be made and the nodes can communicate.

Nodes periodically advertise their presence so that connections can be made with new-found entities, even after the initial discovery period.

Nodes advertise to other nodes when they go offline.

Interfaces

ROS applications typically communicate through interfaces of one of three types: **topics**, **services**, or **actions**:

1. msg: **.msg** files are simple text files that describe the fields of a ROS message.
2. srv: **.srv** files describe a service. They are composed of two parts: a request and a response.
3. action: **.action** files describe actions. They are composed of three parts: a goal, a result, and feedback.

1. Messages

Messages are a way for a ROS 2 node to send data on the network to other ROS nodes, with no response expected. Messages are described and defined in `.msg` files in the `msg/` directory of a ROS package. `.msg` files are composed of two parts: **fields** and **constants**.



For instance, if a ROS 2 node reads temperature data from a sensor, it can then publish that data on the ROS 2 network using a `Temperature` message. Other nodes on the ROS 2 network can subscribe to that data and receive the `Temperature` message

Each **field** consists of a **type** and a **name**, separated by a space:

```
fieldtype1 fieldname1
fieldtype2 fieldname2
fieldtype3 fieldname3
```

For example:

```
int32 my_int
string my_string
```

Field types can be:

- a [built-in-type](#);
- names of Message descriptions defined on their own, such as “`geometry_msgs/PoseStamped`”.

Regarding names, **field names** must be lowercase alphanumeric characters with underscores for separating words.

Default values can be set to any field in the message type. For example:

```
uint8 x 42
int16 y -2000
string full_name "John Doe"
int32[] samples [-200, -100, 0, 100, 200]
```

Each **constant** definition is like a field description with a default value, except that this value can never be changed programatically. This value assignment is indicated by use of an equal ‘=’ sign. For example:

```
int32 X=123
int32 Y=-123
string F00="foo"
string EXAMPLE='bar'
```



Constants names have to be UPPERCASE

2. Services

Services are a request/response communication, where the client (requester) is waiting for the server (responder) to make a short computation and return a result.

Services are described and defined in `.srv` files in the `srv/` directory of a ROS package.

A service description file consists of a request and a response msg type, separated by `---`, as shown below:

```
# request constants
int8 F00=1
int8 BAR=2
# request fields
int8 foobar
another_pkg/AnotherMessage msg
---
# response constants
uint32 SECRET=123456
# response fields
another_pkg/YetAnotherMessage val
CustomMessageDefinedInThisPackage value
uint32 an_integer
```



You cannot embed another service inside of a service.

3. Actions

Actions are a long-running request/response communication, where the action client (requester) is waiting for the action server (the responder) to take some action and return a result.



In contrast to services, actions can be long-running (many seconds or minutes), provide feedback while they are happening, and can be interrupted.

Action definitions have the following form:

```
<request_type> <request_fieldname>
---
<response_type> <response_fieldname>
---
<feedback_type> <feedback_fieldname>
```



The `<request_type>`, `<response_type>`, and `<feedback_type>` follow all of the same rules as the `<type>` for a message. The `<request_fieldname>`, `<response_fieldname>`, and `<feedback_fieldname>` follow all of the same rules as the `<fieldname>` for a message.

Topics

Topics should be used for continuous data streams, like sensor data, robot state, etc.

A publish/subscribe system is one in which there are producers of data (publishers) and consumers of data (subscribers). The **publishers** and **subscribers** know how to contact each other through the concept of a “topic”, which is a common name so that the entites can find each other. Any publishers and subscribers that are on the same topic name can directly communicate with each other.

When data is published to the topic by any of the publishers, all subscribers in the system will receive the data. This system is also known as a “bus”, since it somewhat resembles a device bus from electrical engineering.



ROS 2 is “*anonymous*”. This means that when a subscriber gets a piece of data, it doesn’t generally know or care which publisher originally sent it (though it can find out if it wants). The benefit to this architecture is that publishers and subscribers can be swapped out at will without affecting the rest of the system.



The publish/subscribe system is “*strongly-typed*”. This means that the data that is published and subscribed is described by a message type, and the publisher and subscriber must agree on the message type.

Services

A **service** refers to a remote procedure call. In other words, a node can make a remote procedure call to another node which will do a computation and return a result.

In ROS 2, services are expected to return quickly, as the client is generally waiting on the result. Services should never be used for longer running processes. If you have a service that will be doing a long-running computation, consider using an action instead.

Services are identified by a service name, which looks much like a topic name.

A service consists of two parts: the **service server** and the **service client**.

- **Service server**: is the entity that will accept a remote procedure request, and perform some computation on it.



There should only ever be one service server per service name.

- **Service client**: is an entity that will request a remote service server to perform a computation on its behalf.



Unlike the service server, there can be arbitrary numbers of service clients using the same service name.

Actions

An **action** refers to a long-running remote procedure call with feedback and the ability to cancel or preempt the goal.



For instance, the high-level state machine running a robot may call an action to tell the navigation subsystem to travel to a waypoint, which may take several seconds (or minutes) to do. Along the way, the navigation subsystem can provide feedback on how far along it is, and the high-level state machine has the option to cancel or preempt the travel to that waypoint.

Actions are identified by an action name, which looks much like a topic name.

An action consists of two parts: the **action server** and the **action client**.

- **Action server**: is the entity that will accept the remote procedure request and perform some procedure on it. It is also responsible for sending out feedback as the action progresses and should react to cancellation/preemption requests.



There should only ever be one action server per action name.

- **Action client**: is an entity that will request a remote action server to perform a procedure on its behalf.



Unlike the action server, there can be arbitrary numbers of action clients using the same action name.

Parameters

Parameters are associated with individual nodes. Parameters are used to configure nodes at startup (and during runtime), without changing the code. The lifetime of a parameter is tied to the lifetime of the node.

Parameters are addressed by **node name**, **node namespace**, **parameter name**, and **parameter namespace** (providing a parameter namespace is optional).



For example, let's consider a node with the name **robot_controller** and a parameter named **max_speed**. If the parameter is within the namespace of that node, it might be identified as **robot_controller/max_speed**. This enables having parameters with the same name but distinct to different nodes by utilizing the node's namespace.

Furthermore, the **parameter namespace** is an additional subdivision within the node's namespace, allowing for further organization and hierarchy. For example, a parameter could be **robot_controller/motion_parameters/max_speed**. In this case, **motion_parameters** is a parameter namespace that helps group parameters related to the robot's motion parameters under a specific category.

Each parameter consists of a **key**, a **value**, and a **descriptor**.

- The **key** is a string;
- The **value** is one of the following types: `bool`, `int64`, `float64`, `string`, `byte[]`, `bool[]`, `int64[]`, `float64[]` or `string[]`.
- By default all **descriptors** are empty, but can contain parameter descriptions, value ranges, type information, and additional constraints.

By default, a node needs to declare all of the parameters that it will accept during its lifetime.



For some types of nodes, not all of the parameters will be known ahead of time. In these cases, the node can be instantiated with `allow_undeclared_parameters` set to `true`, which will allow parameters to be get and set on the node even if they haven't been declared.

Each parameter on a ROS 2 node has one of the pre-defined parameter types (`bool`, `int64`, `float64`, `string`, `byte[]`, `bool[]`, `int64[]`, `float64[]` or `string[]`).



If a parameter needs to be multiple different types, and the code using the parameter can handle it, this default behavior can be changed. When the parameter is declared, it should be declared using a `ParameterDescriptor` with the `dynamic_typing` member variable set to `true`.

A ROS 2 node can register two different types of callbacks to be informed when changes are happening to parameters. Both of the callbacks are optional.

- **set parameters** callback:
 - Can be set by calling `add_on_set_parameters_callback` from the node API;
 - The callback is passed a list of immutable `Parameter` objects, and returns an `rcl_interfaces/msg/SetParametersResult`;
 - The main purpose of this callback is to give the user the ability to inspect the upcoming change to the parameter and explicitly reject the change.



Let's consider a ROS 2 node that manages the speed of a robot as a parameter. The **set parameters** callback could be used to control and potentially reject sudden changes to the robot's speed that might cause issues.

```
import rclpy
from rcl_interfaces.msg import SetParametersResult
from my_robot_control import RobotControl

def set_parameter_callback(parameters):
    for param in parameters:
        if param.name == 'robot_speed' and param.value > 5.0:
            # Reject the change if the new speed is too high
            return SetParametersResult(successful=False, reason='Speed
exceeds limit')
        # Accept the change for other parameters
        return SetParametersResult(successful=True)

def main():
```

```

rclpy.init()
node = rclpy.create_node('robot_controller')

# Register the "set parameter" callback
node.add_on_set_parameters_callback(set_parameter_callback)

robot_control = RobotControl()

try:
    rclpy.spin(node)
except KeyboardInterrupt:
    pass
finally:
    node.destroy_node()
    rclpy.shutdown()

if __name__ == '__main__':
    main()

```

In this example, if someone attempts to set the robot's speed above 5.0 via parameters, the callback will reject the change and provide a specific reason.

- **on parameter event** callback:
 - Can be set by calling `on_parameter_event` from the parameter client APIs.
 - The callback is passed an `rcl_interfaces/msg/ParameterEvent` object, and returns nothing. This callback will be called after all parameters in the input event have been declared, changed, or deleted.
 - The main purpose of this callback is to give the user the ability to react to changes from parameters that have successfully been accepted.



Suppose we want to log an event every time the robot's configuration is modified. We could use an **on parameter event** callback to capture this event and take corresponding actions.

```

import rclpy
from rcl_interfaces.msg import ParameterEvent
from my_robot_control import RobotControl

def on_parameter_event_callback(parameter_event):
    if 'robot_config' in parameter_event.changed_parameters:
        # Perform specific actions when the robot's configuration
        changes
        print("The robot's configuration has been modified:",
              parameter_event.changed_parameters['robot_config'])

def main():
    rclpy.init()
    node = rclpy.create_node('robot_monitor')

    # Register the "on parameter event" callback
    node.on_parameter_event(on_parameter_event_callback)

```

```
try:
    rclpy.spin(node)
except KeyboardInterrupt:
    pass
finally:
    node.destroy_node()
    rclpy.shutdown()

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

In this example, every time the robot's configuration changes, the **on parameter event** callback is invoked, allowing the node to perform specific actions in response to this event.

In addition, regarding parameters, there are other concepts to be aware of:

1. **Interacting with parameters:** [External processes](#) can perform parameter operations via parameter services that are created by default when a node is instantiated;
2. **Setting initial parameter values when running a node:** Initial parameter values can be set when running the node either through individual command-line arguments, or through YAML files;
3. **Setting initial parameter values when launching nodes:** Initial parameter values can also be set when running the node through the ROS 2 launch facility;
4. **Manipulating parameter values at runtime:** The `ros2 param` command is the general way to interact with parameters for nodes that are already running. `ros2 param` uses the [parameter service API](#) to perform the various operations.

Introspection with command line tools

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

To see all available sub-commands run:

```
ros2 --help
```

Examples of sub-commands that are available include:

- `action`: Introspect/interact with ROS actions;
- `bag`: Record/play a rosbag;
- `component`: Manage component containers;
- `daemon`: Introspect/configure the ROS 2 daemon;
- `doctor`: Check ROS setup for potential issues;
- `interface`: Show information about ROS interfaces;
- `launch`: Run/introspect a launch file;
- `node`: Introspect ROS nodes;
- `param`: Introspect/configure parameters on a node;

- **pkg**: Introspect ROS packages;
- **run**: Run ROS nodes;
- **security**: Configure security settings;
- **service**: Introspect/call ROS services;
- **test**: Run a ROS launch test;
- **topic**: Introspect/publish ROS topics.

Launch

A ROS 2 system typically consists of many nodes running across many different processes. While it is possible to run each of these nodes separately, it gets cumbersome quite quickly.

The **launch system** in ROS 2 is meant to automate the running of many nodes with a single command. It helps the user describe the configuration of their system and then executes it as described. The configuration of the system includes what programs to run, where to run them, what arguments to pass them.

All of the above is specified in a **launch file**, which can be written in Python, XML, or YAML. This launch file can then be run using the **ros2 launch** command, and all of the nodes specified will be run.

For example:

```
from launch import LaunchDescription
from launch_ros.actions import Node

def generate_launch_description():
    return LaunchDescription([
        Node(
            package='demo_nodes_cpp',
            executable='talker',
            output="screen",          # To see the output of the node
            parameters=[{
                'serial_port': '/dev/ttyUSB0',
                'use_sim_time': True,   # To set use_sim_time parameter
            }],
            remapping=[
                ('\scan', '\scan1'),
            ]
        ),
        Node(
            package='demo_nodes_cpp',
            executable='listener',
            namespace='/scanner2',
            parameters=[{
                'serial_port': '/dev/ttyUSB1',
            }],
        )
    ])

```

Client Libraries

Client libraries are the APIs that allow users to implement their ROS 2 code. Using client libraries, users gain access to ROS 2 concepts such as **nodes**, **topics**, **services**, etc.



Nodes written using different client libraries are able to share messages with each other because all client libraries implement code generators that provide users with the capability to interact with ROS 2 interface files in the respective language.

1. The rclcpp package

The **ROS Client Library for C++ (rclcpp)** is the user facing, C++ idiomatic interface which provides all of the ROS client functionality like creating nodes, publishers, and subscriptions.

rclcpp makes use of all the features of C++ and C++17 to make the interface as easy to use as possible, but since it reuses the implementation in **rcl** it is able maintain a consistent behavior with the other client libraries that use the **rcl** API.

2. The rclpy package

The **ROS Client Library for Python (rclpy)** is the Python counterpart to the C++ client library. Like the C++ client library, **rclpy** also builds on top of the **rcl** C API for its implementation.

By using the **rcl** API in the implementation, it stays consistent with the other client libraries in terms of feature parity and behavior.

3. Common functionality: rcl

Most of the functionality found in a client library is not specific to the programming language of the client library.

Client libraries make use of a **common core ROS Client Library (RCL)** interface that implements logic and behavior of ROS concepts that is not language-specific. As a result, client libraries only need to wrap the common functionality in the **RCL** with foreign function interfaces. This keeps client libraries thinner and easier to develop. For this reason the common **RCL** functionality is exposed with C interfaces as the C language is typically the easiest language for client libraries to wrap.

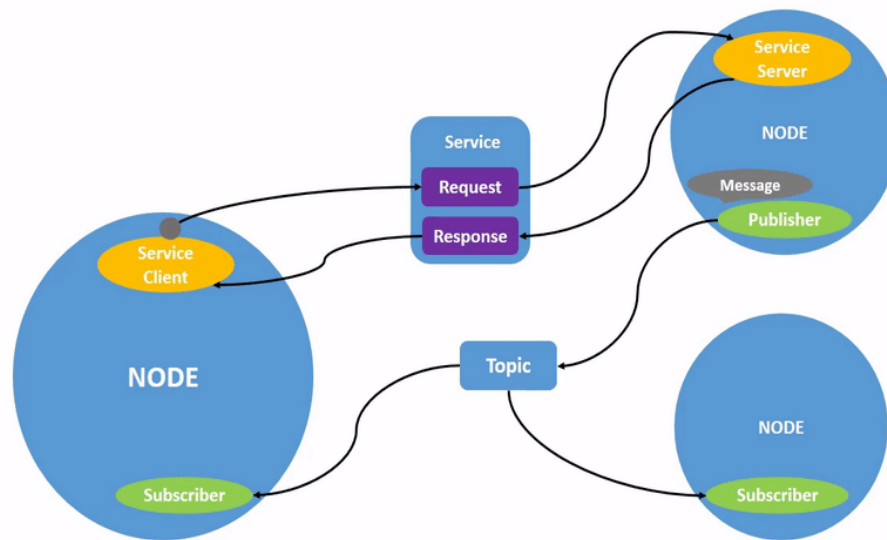


Client library concepts that require language-specific features/properties are not implemented in the RCL but instead are implemented in each client library. For example, threading models used by “spin” functions will have implementations that are specific to the language of the client library.

Command Line Interface Tools

Understanding Nodes

Each node in ROS should be responsible for a single, modular purpose, e.g. controlling the wheel motors or publishing the sensor data from a laser range-finder. Each node can send and receive data from other nodes via topics, services, actions, or parameters.



[Animated image here](#)

1. `ros2 run`

The command **ros2 run** launches an executable from a package.

```
ros2 run <package_name> <executable_name>
```



If you want to run a node with a parameter, you can use the `---ros-args -p` option. For example:

```
ros2 run <package_name> <executable_name> --ros-args -p
my_param:=my_value
#ros2 run cbf-stl_pkg input_node --ros-args -p use_sim_time:=true
```

2. `ros2 node list`

ros2 node list will show you the names of all running nodes. This is especially useful when you want to interact with a node, or when you have a system running many nodes and need to keep track of them.

```
ros2 node list
```

3. Remapping

Remapping allows you to reassign default node properties, like node name, topic names, service names, etc., to custom values.

- Names within a node (e.g. topics/services) can be remapped using the syntax:

```
ros2 run <package_name> <executable_name> --ros-args -r <old_name>:=<br><new_name>
```

- The name/namespace of the node itself can be remapped using the syntax:

```
# remap node name
ros2 run <package_name> <executable_name> --ros-args -r __node:=<br><new_node_name>

# remap node namespace
ros2 run <package_name> <executable_name> --ros-args -r __ns:=<br><new_node_namespace>
```

4. ros2 node info

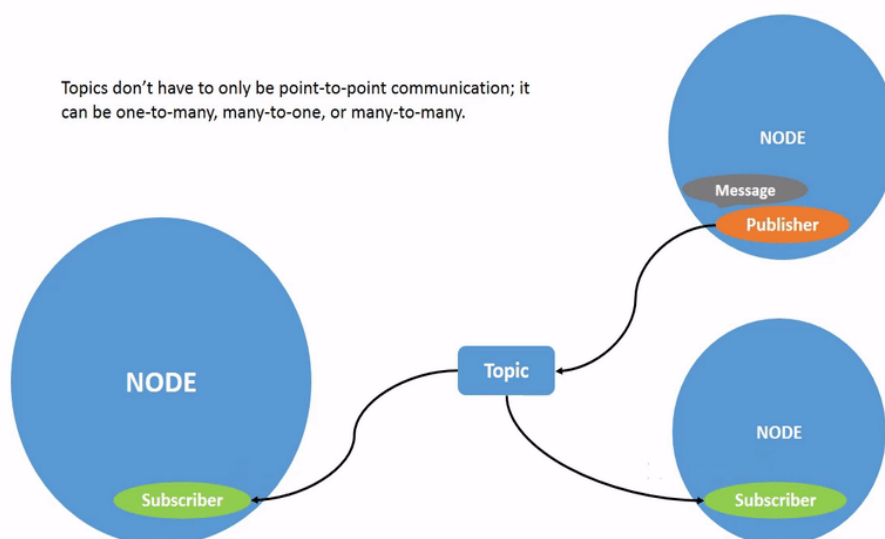
Now that you know the names of your nodes, you can access more information about them with:

```
ros2 node info <node_name>
```

ros2 node info returns a list of subscribers, publishers, services, and actions.

Understanding Topics

ROS 2 breaks complex systems down into many modular nodes. Topics are a vital element of the ROS graph that act as a bus for nodes to exchange messages.



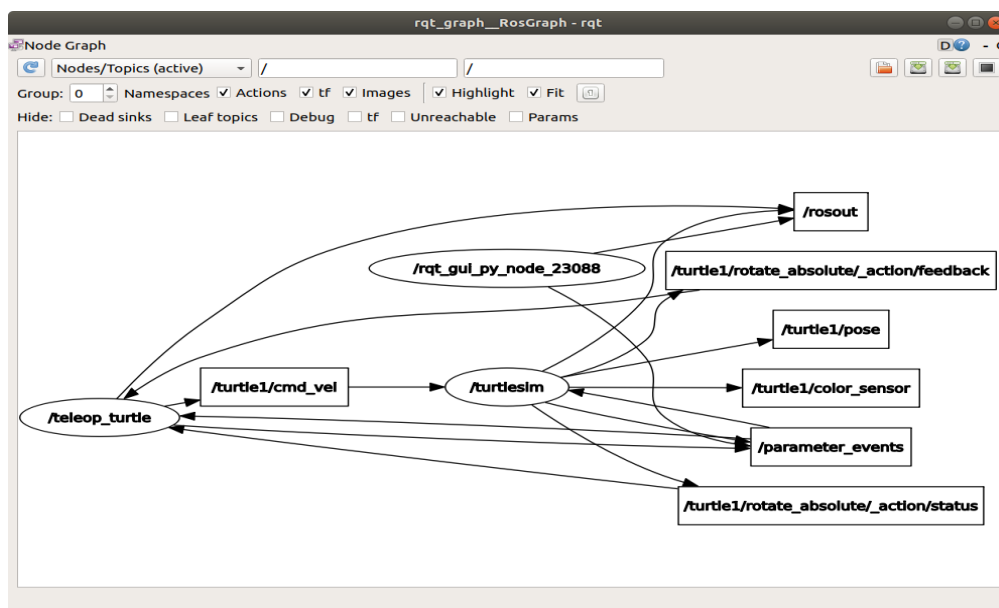
[Animated image here](#)

A node may publish data to any number of topics and simultaneously have subscriptions to any number of topics. Topics are one of the main ways in which data is moved between nodes and therefore between different parts of the system.

1. rqt-graph

rqt_graph is used to visualize the changing nodes and topics, as well as the connections between them. To run rqt_graph, open a new terminal and enter the command:

```
rqt_graph
```



2. ros2 topic list

Running the **ros2 topic list** command in a new terminal:

```
ros2 topic list
```

will return a list of all the topics currently active in the system.



ros2 topic list -t: will return the same list of topics, this time with the topic type appended in brackets.

3. ros2 topic echo

To see the data being published on a topic, use:

```
ros2 topic echo <topic_name>
```

4. ros2 interface show

Nodes send data over topics using messages. Publishers and subscribers must send and receive the same type of message to communicate.

We can run:

```
ros2 interface show <msg type>
```

to learn the details of the msg type. Which will return something like:

```
Vector3  linear
         float64 x
         float64 y
         float64 z
Vector3  angular
         float64 x
         float64 y
         float64 z
```

5. ros2 topic pub

It is possible to publish data into a topic directly from the command line using:

```
ros2 topic pub <topic_name> <msg_type> '<args>'
```



It's important to note that '**<args>**' needs to be input in YAML syntax.

Input the full command like:

```
ros2 topic pub --once /turtle1/cmd_vel geometry_msgs/msg/Twist "{linear:
{x: 2.0, y: 0.0, z: 0.0}, angular: {x: 0.0, y: 0.0, z: 1.8}}"
ros2 topic pub /emotion std_msgs/msg/String "{data: 'Hello from terminal'}"
ros2 topic pub /enable std_msgs/msg/Bool "{data: True}"
```



--once is an optional argument meaning “publish one message then exit”.

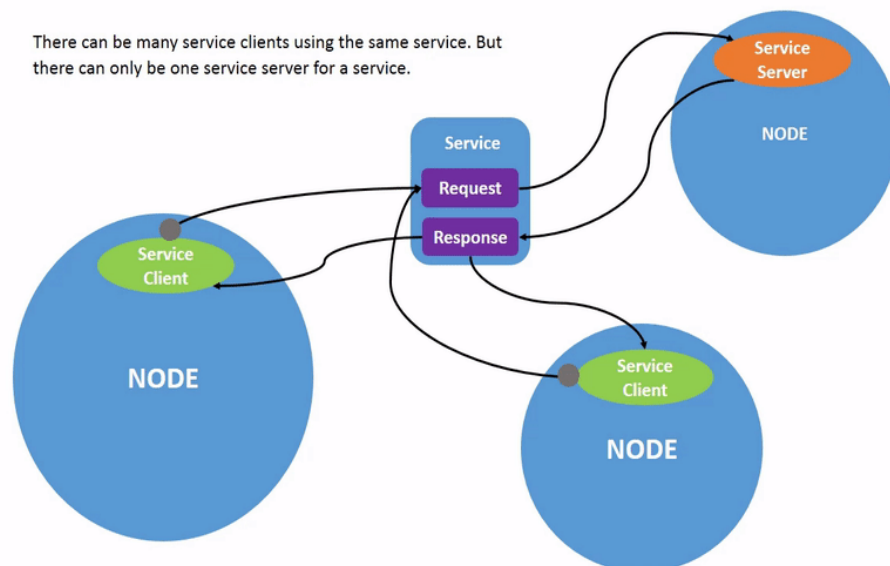
6. ros2 topic hz

ros2 topic hz is a useful tool for measuring the publishing rate of a topic. It will return the publishing rate of a topic in hertz (Hz), i.e. the number of messages published per second.

```
ros2 topic hz <topic_name>
```

Understanding Services

Services are another method of communication for nodes in the ROS graph. Services are based on a call-and-response model versus the publisher-subscriber model of topics. While topics allow nodes to subscribe to data streams and get continual updates, services only provide data when they are specifically called by a client.



[Animated image here](#)

1. ros2 service list

Running the **ros2 service list** command in a new terminal will return a list of all the services currently active in the system.

```
ros2 service list
```

2. ros2 service type

Services have types that describe how the request and response data of a service is structured. Service types are defined similarly to topic types, except service types have two parts: one message for the request and another for the response.

To find out the type of a service, use the command:

```
ros2 service type <service_name>
```



To see the types of all the active services at the same time, you can append the `--show-types` option, abbreviated as `-t`, to the list command:

```
ros2 service list -t
```

3. ros2 service find

If you want to find all the services of a specific type, you can use the command:

```
ros2 service find <type_name>
```

4. ros2 interface show

To see the structure of a service type, use the command:

```
ros2 interface show <service_type>
```

Which will return something like:

```
float32 x
float32 y
float32 theta
string name
---
string name
```

The information above the `---` line tells us the structure of the request data, and the information below the `---` line tells us the structure of the response data.

5. ros2 service call

It is possible to call a service using:

```
ros2 service call <service_name> <service_type> <arguments>
```

Where the `<arguments>` part is optional.

Understanding Parameters

A **parameter** is a configuration value of a node. You can think of parameters as node settings. A node can store parameters as integers, floats, booleans, strings, and lists. In ROS 2, each node maintains its own parameters.

1. ros2 param list

To see the parameters belonging to your nodes, open a new terminal and enter the command:

```
ros2 param list
```



Every node has the parameter `use_sim_time`.

2. ros2 param get

To display the type and current value of a parameter, use the command:

```
ros2 param get <node_name> <parameter_name>
```

3. ros2 param set

To change a parameter's value at runtime, use the command:

```
ros2 param set <node_name> <parameter_name> <value>
```



Setting parameters with the `set` command will only change them in your current session, not permanently. However, you can save your settings and reload them the next time you start a node.

4. ros2 param dump

You can view all of a node's current parameter values by using the command:

```
ros2 param dump <node_name>
```

It is possible to save your current configuration of the node's parameters to a `.yaml` file using the command:

```
ros2 param dump <node_name> > <file_name>.yaml
```



Dumping parameters comes in handy if you want to reload the node with the same parameters in the future.

5. ros2 param load

You can load parameters from a file to a currently running node using the command:

```
ros2 param load <node_name> <parameter_file>
```

6. Load parameter file on node startup

To start the same node using your saved parameter values, use:

```
ros2 run <package_name> <executable_name> --ros-args --params-file  
<file_name>
```

Understanding Actions

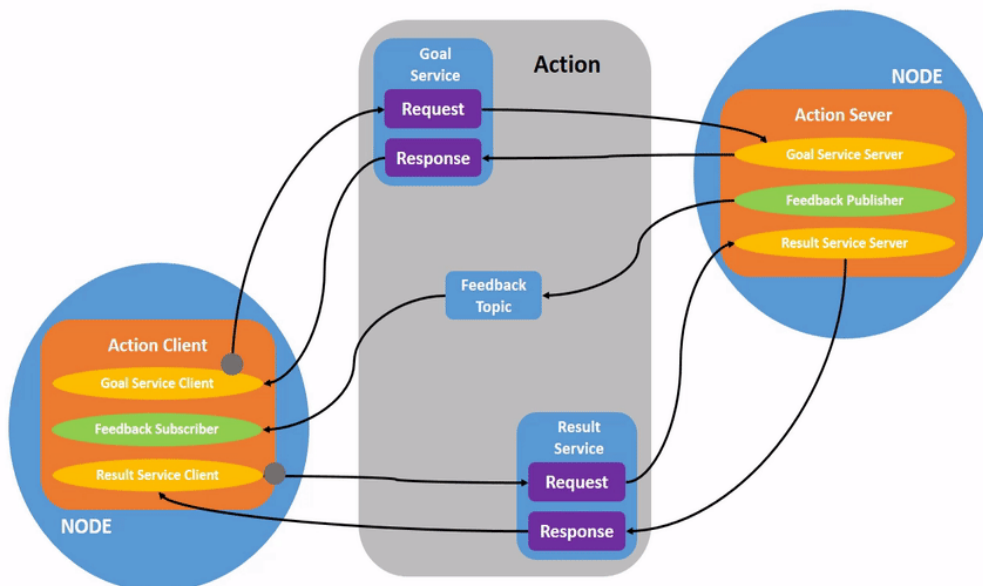
Actions are one of the communication types in ROS 2 and are intended for long running tasks. They consist of three parts: a goal, feedback, and a result.

Actions are built on topics and services. Their functionality is similar to services, except actions can be canceled. They also provide steady feedback, as opposed to services which return a single response.

Actions use a client-server model, similar to the publisher-subscriber model. An “action client” node sends a goal to an “action server” node that acknowledges the goal and returns a stream of feedback and a result.



A robot system would likely use actions for navigation. An action goal could tell a robot to travel to a position. While the robot navigates to the position, it can send updates along the way (i.e. feedback), and then a final result message once it's reached its destination.



[Animated image here](#)

1. ros2 node info

To see the list of actions a node provides, open a new terminal and run the command:

```
ros2 node info <node_name>
```

Which will return a list of node's subscribers, publishers, services, action servers and action clients. For example:

```
/turtlesim
Subscribers:
  /parameter_events: rcl_interfaces/msg/ParameterEvent
  /turtle1/cmd_vel: geometry_msgs/msg/Twist
Publishers:
  /parameter_events: rcl_interfaces/msg/ParameterEvent
  /rosout: rcl_interfaces/msg/Log
  /turtle1/color_sensor: turtlesim/msg/Color
  /turtle1/pose: turtlesim/msg/Pose
Service Servers:
  /clear: std_srvs/srv/Empty
  /kill: turtlesim/srv/Kill
  /reset: std_srvs/srv/Empty
  /spawn: turtlesim/srv/Spawn
Service Clients:

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

2. ros2 action list

To identify all the actions in the ROS graph, run the command:

```
ros2 action list
```



To find action's type, run the command:

```
ros2 action list -t
```

3. ros2 action info

You can further introspect the action with the command:


```
ros2 action info <action_type>
```

Which will return something like:

```
Action: /turtle1/rotate_absolute
Action clients: 1
    /teleop_turtle
Action servers: 1
    /turtlesim
```

4. ros2 interface show

To see the structure of an action type, run the command:

```
ros2 interface show <action_type>
```

Which will return something like:


```
#The desired heading in radians
float32 theta
---
#The angular displacement in radians to the starting position
float32 delta
---
#The remaining rotation in radians
float32 remaining
```

The section of this message above the first `---` is the structure (data type and name) of the goal request. The next section is the structure of the result. The last section is the structure of the feedback.

5. ros2 action send_goal

To send a goal to an action server, use the command:

```
ros2 action send_goal <action_name> <action_type> <values>
```

 The `<values>` need to be in YAML syntax.

For example:

```
ros2 action send_goal /turtle1/rotate_absolute  
turtlesim/action/RotateAbsolute "{theta: 1.57}"
```



To see the feedback of the goal, add `--feedback` to the `ros2 action send_goal` command. In this way, you will continue to receive feedback, until the goal is complete.

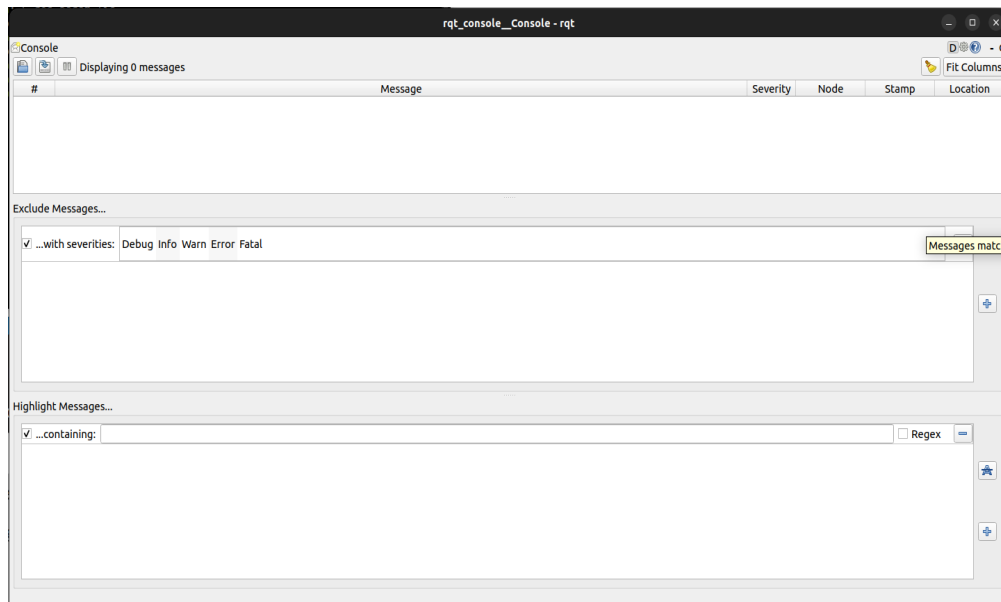
rqt_console

rqt_console is a GUI tool used to introspect log messages in ROS 2. Typically, log messages show up in your terminal. With `rqt_console`, you can collect those messages over time, view them closely and in a more organized manner, filter them, save them and even reload the saved files to introspect at a different time.

Nodes use logs to output messages concerning events and status in a variety of ways. Their content is usually informational, for the sake of the user.

Start `rqt_console` in a new terminal with the following command:

```
ros2 run rqt_console rqt_console
```



The first section of the console is where log messages from your system will display.

In the middle you have the option to filter messages by excluding severity levels. You can also add more exclusion filters using the plus-sign button to the right.

The bottom section is for highlighting messages that include a string you input. You can add more filters to this section as well.


1. Logger Levels

ROS 2's logger levels are ordered by severity:

- **Fatal** messages indicate the system is going to terminate to try to protect itself from detriment.
- **Error** messages indicate significant issues that won't necessarily damage the system, but are preventing it from functioning properly.
- **Warn** messages indicate unexpected activity or non-ideal results that might represent a deeper issue, but don't harm functionality outright.
- **Info** messages indicate event and status updates that serve as a visual verification that the system is running as expected.
- **Debug** messages detail the entire step-by-step process of the system execution.

ros2 bag

ros2 bag is a command line tool for recording data published on topics in your system. It accumulates the data passed on any number of topics and saves it in a database. You can then replay the data to reproduce the results of your tests and experiments. Recording topics is also a great way to share your work and allow others to recreate it.

 You should have **ros2 bag** installed as a part of your regular ROS 2 setup. If you installed ROS from Debian packages on Linux and your system doesn't recognize the command, install it like so:

```
sudo apt-get install ros-humble-ros2bag \ros-humble-rosbag2-storage-  
default-plugins
```

1. Setup


To record data, you need to create a directory to store the database file. Open a new terminal and run:

```
mkdir bag_files  
cd bag_files
```

2. ros2 bag record

ros2 bag can only record data from published messages in topics. To record data from a topic, open a new terminal and run the command:

```
ros2 bag record <topic_name>
```

 Before running this command on your chosen topic, open a new terminal and move into the **bag_files** directory you created earlier, because the **rosvag** file will save in the directory where you run it.

Press **Ctrl+C** to stop recording.

3. ros2 bag info

You can see details about your recording by running:

```
ros2 bag info <bag_file_name>
```

4. ros2 bag play

To replay the data from a bag file, open a new terminal and run the command:

```
ros2 bag play <bag_file_name>
```

Verification and installation packages

All the available packages can be found at the following [link](#).

Verify all your installed packages:

```
sudo apt list --installed
```

Verify that a specific package is installed:

```
sudo apt list --installed | grep <packageName>
```

Install a specific package:

```
sudo apt install ros-humble-<packageName>
```

Clock

The **clock** is a fundamental part of ROS 2. It is used to provide a common time reference for all nodes in the system. The clock is used to timestamp messages, log messages, and to schedule timers. There are different types of clocks, in C++ you can use the following part of code to check the differences:

```
RCLCPP_INFO_STREAM(get_logger(), "now(): " << now().seconds());  
RCLCPP_INFO_STREAM(get_logger(), "rclcpp::Clock{}.now(): " <<  
rclcpp::Clock{}.now().seconds());  
RCLCPP_INFO_STREAM(get_logger(), "rclcpp::Clock{RCL_ROS_TIME}.now(): " <<  
rclcpp::Clock{RCL_ROS_TIME}.now().seconds());
```

```
RCLCPP_INFO_STREAM(get_logger(), "rclcpp::Clock{RCL_SYSTEM_TIME}.now(): "  
<< rclcpp::Clock{RCL_SYSTEM_TIME}.now().seconds());  
RCLCPP_INFO_STREAM(get_logger(), "rclcpp::Clock{RCL_STEADY_TIME}.now(): "  
<< rclcpp::Clock{RCL_STEADY_TIME}.now().seconds());
```

If you want to use the simulation clock you need to enable `use_sim_time` parameter. You can do this by running the following command:

```
ros2 run <package_name> <executable_name> --ros-args -p use_sim_time:=true
```

or adding it to the launch file.

Furthermore, if you want to use the simulation clock you need to use the following timer in C++:

```
timer_compute_ = rclcpp::create_timer(this, this->get_clock(),  
std::chrono::milliseconds(20), std::bind(&Cb_f_stl::timer_compute_callback,  
this));  
//instead of:  
//timer_compute_ = this->create_wall_timer(20ms,  
std::bind(&Cb_f_stl::timer_compute_callback, this));
```

Workspace and Packages

A **package** is a way for us to group together a whole lot of related files, so that we can use them again in a different project later or just to keep them grouped. You can put anything you want in a package (nodes, launch, robot models, documentation, etc).



A package has a list of other packages that it depends on packages that it needs to work. These are called **dependencies**.

When we work on ROS projects, we create a special folder called a **workspace** to keep all of our packages in. Generally makes sense to have one workspace per project and inside this are going to be all the packages that we need for that project.

Create a Workspace

Firstable, you need to create your workspace. You can create the project folder wherever you want, but it is recommended to create it in the **Home** folder and to put **_ws** at the end of the name.

Create, for example, a folder called **test_ws** and inside it create another folder called **src**.

```
mkdir -p <workspace_name>_ws/src
#mkdir -p test_ws/src
```

Then go into the **test_ws** folder and do:

```
colcon build --symlink-install
```



colcon build will create the following directories:

- The **build** directory will be where intermediate files are stored. For each package a subfolder will be created in which e.g. CMake is being invoked.
- The **install** directory is where each package will be installed to. By default each package will be installed into a separate subdirectory.
- The **log** directory contains various logging information about each colcon invocation.

Create a Package

A single **workspace** can contain as many **packages** as you want, each in their own folder. You can also have packages of different build types in one workspace (CMake, Python, etc.). You cannot have nested packages.

Best practice is to have a **src** folder within your workspace, and to create your packages in there. For convenience, you can use the tool **ros2 pkg create** to create a new package based on a template.

```
#for c++ package
cd test_ws/src
ros2 pkg create --build-type ament_cmake --license Apache-2.0 --node-name
<node_name>_node <package_name>_pkg

#for python package
cd test_ws/src
ros2 pkg create --build-type ament_python --license Apache-2.0 --node-name
<node_name>_node <package_name>_pkg
```

Please, read [C++ package](#) and [Python package](#) before to create a package.

1. C++ package

The simplest possible package may have a file structure that looks like:

- `CMakeLists.txt` file that describes how to build the code within the package;
- `include/<package_name>` directory containing the public headers for the package;
- `package.xml` file containing meta information about the package;
- `src` directory containing the source code for the package.

2. Python package

The simplest possible package may have a file structure that looks like:


- `package.xml` file containing meta information about the package;
- `resource/<package_name>` marker file for the package;
- `setup.cfg` is required when a package has executables, so `ros2 run` can find them. This file simply telling `setuptools` to put your executables in `lib`, because `ros2 run` will look for them there.;
- `setup.py` containing instructions for how to install the package;
- `<package_name>` a directory with the same name as your package, used by ROS 2 tools to find your package, contains `__init__.py` and your nodes.

Costumize package.xml

You may have noticed in the return message, after creating your package, that the fields `description` and `license` contain `TODO` notes:

- Input your `name` and `email` on the `maintainer` line;
- Edit the `description` line to summarize the package;
- Update the `license` line using `Apache License 2.0`.

Below the license tag, you will see some tag names ending with `_depend`. This is where your `package.xml` would list its dependencies corresponding to your **node's include statements**.

 Only for Python packages: The `setup.py` file contains the same `description`, `maintainer` and `license` fields as `package.xml`, so you need to set those as well. They need to match exactly in both files. The version and name (`package_name`) also need to match exactly, and should be automatically populated in both files.

Add dependencies

After that you have written your code you need to add the dependencies of your package. *Dependencies_* corresponding to your **node's include statements**.

C++ package

- In `package.xml`, add new lines after `<buildtool_depend>ament_cmake</buildtool_depend>` with your dependencies. For example:

```
<buildtool_depend>ament_cmake</buildtool_depend>

<depend>rclcpp</depend>
<depend>std_msgs</depend>
```

- In `CMakeLists.txt`, add new lines after `find_package(ament_cmake REQUIRED)` with your dependencies. For example:

```
find_package(ament_cmake REQUIRED)

find_package(rclcpp REQUIRED)
find_package(std_msgs REQUIRED)
```

- After that, in `CMakeLists.txt`, add the executable and name it `talker` so you can run your node using `ros2 run`:

```
add_executable(talker src/publisher_member_function.cpp)
ament_target_dependencies(talker rclcpp std_msgs)

#add_executable(listener src/subscriber_member_function.cpp)
#ament_target_dependencies(listener rclcpp std_msgs)
```

- Finally, in `CMakeLists.txt`, add the `install(TARGETS...)` section so `ros2 run` can find your executable:


```
install(TARGETS
  talker
  #listener
  DESTINATION lib/${PROJECT_NAME})
```

Python package

- In `package.xml`, add new lines after `<buildtool_depend>ament_python</buildtool_depend>` with your dependencies. For example:

```
<buildtool_depend>ament_python</buildtool_depend>

<depend>rcpp</depend>
<depend>std_msgs</depend>
```

- In `setup.py`, add an entrypoint. This is required when a package has executables, so `ros2 run` can find them. For example:

```
entry_points={
    'console_scripts': [
        '<executable_name> = <package_name>.<node_name>:main',
        #'talker = py_pubsub.publisher_member_function:main'
        #'listener = py_pubsub.subscriber_member_function:main'
    ],
}
```

- Check in `setup.cfg` that the content is populated like so:

```
[develop]
script-dir=$base/lib/<package_name>
[install]
install-scripts=$base/lib/<package_name>
```

Build Workspace

Before building the workspace, you need to resolve the package dependencies. To do this, you can use the tool `rosdep`.

```
cd test_ws
rosdep install -i --from-path src --roscat humble -y
```

If you already have all your dependencies, the console will return:

```
#All required rosdeps installed successfully
```

Then, you can build the workspace:

```
colcon build --symlink-install
```



Some notes:

- If you do not want to build a specific package place an empty file named `COLCON_IGNORE` inside the package folder;
- If you want to build a specific package, you can use `colcon build --packages-select <my_package>`.

Add personal / external library

If you want to add a personal or external library to your package, you need to add the cpp files in the package folder and the header files in the include folder in a dedicated folder.

For example, you should have the following structure:

```
<your_package>_pkg
├── include
│   ├── <package_name>
│   │   └── <library_name>
│   │       ├── <first_file_library>.h
│   │       ├── <second_file_library>.h
│   │       └── ...
├── src
│   ├── <your_node>_node.cpp
│   ├── <library_name>
│   │   ├── <first_file_library>.cpp
│   │   ├── <second_file_library>.cpp
│   │   └── ...
├── CMakeLists.txt
└── package.xml
```

Then, in the `CMakeLists.txt` file, you need to add the following lines:

```
# <-- Insert here your dependences -->
...
...
# <-- _____ -->
#*****
```

```
# Building your library
add_library(<library_name>
  src/<library_name>/<first_file_library>.cpp
  src/<library_name>/<second_file_library>.cpp
  src/<library_name>/<...>.cpp
)

# Include headers
include_directories(include/<library_name>)
#*****

# Insert here your executables and ament_target_dependencies
add_executable(<your_node>_node src/<your_node>_node.cpp)
ament_target_dependencies(<your_node>_node rclcpp rclcpp_action
  rclcpp_components std_msgs ...)
target_link_libraries (<your_node>_node <library_name>)
```

Source workspace

Remember that every time you open a new terminal and you want run your node, you need to source the workspace. To do this, run the following command:

```
source <workspace_name>_ws/install/setup.bash
```

otherwise you can add the following line to the `.bashrc` file:

```
source ~/<workspace_name>_ws/install/setup.bash
```

and everytime you open a new terminal, the workspace will be sourced.

Create a Publisher and Subscriber

C++ Publisher and Subscriber

1. Create a C++ package

Use the following command to create a new C++ package:

```
ros2 pkg create --build-type ament_cmake --license Apache-2.0 --node-name
<node_name>_node <package_name>_pkg
```

2. Create a C++ publisher and Subscriber

Go into `template_ws` in which you can find a `robot_pkg` package. His node, `robot_node`, is a simple publisher and subscriber. You can use it as a template to create your own publisher and subscriber.



If you want to know the timestamp of the message, you can use the following code in the subscriber callback function:

```
int old_sec = 0;
int old_nsec = 0;
void callback(const std_msgs::msg::String::SharedPtr msg){
    int new_sec = msg->header.stamp.sec;
    int new_nsec = msg->header.stamp.nanosec;
    RCLCPP_INFO(rclcpp::get_logger("SUBSCRIBER ODOM"), "Received at: %d
sec", msg->header.stamp.sec);
    RCLCPP_INFO(rclcpp::get_logger("SUBSCRIBER ODOM"), "Received at: %d
nanosec", msg->header.stamp.nanosec);
    RCLCPP_INFO(rclcpp::get_logger("SUBSCRIBER ODOM"), "Time difference:
%d sec and %d nanosec", diff_sec, diff_nsec);
    old_sec = new_sec;
    old_nsec = new_nsec;
}
```

3. Add dependencies

You need to add the dependencies of your package in the `package.xml` and in the `CMakeLists.txt` files. Go into `template_ws` in which you can find a `robot_pkg` package. His `package.xml` and `CMakeLists.txt` files are a good example to add the dependencies.

Python Publisher and Subscriber

1. Create a Python package

Use the following command to create a new Python package:

```
ros2 pkg create --build-type ament_python --license Apache-2.0 --node-name  
<node_name>_node <package_name>_pkg
```

2. Create a Python publisher and Subscriber

Go into `template_ws` in which you can find a `camera_pkg` package. His node, `camera_node`, is a simple publisher and subscriber. You can use it as a template to create your own publisher and subscriber.

3. Add dependencies

You need to add the dependencies of your package in the `package.xml` and in the `setup.py` files. Go into `template_ws` in which you can find a `camera_pkg` package. His `package.xml` and `setup.py` files are a good example to add the dependencies.

Create my_interfaces package

It is possible to create your custom `.msg` and `.srv` files, and then utilizing them in a separate package. Both packages should be in the same workspace. Go into the `src` folder of your workspace and create a new package called `my_interfaces`:

```
ros2 pkg create --build-type ament_cmake my_interfaces --license Apache-2.0
```

It is good practice to keep `.msg` and `.srv` files in their own directories within a package. Inside `my_interfaces` workspace, create two directories called `msg` and `srv`:

```
mkdir msg  
  
mkdir srv
```

1. Create custom msg definitions

For example, create a file called `Num.msg` in the `msg` directory, declaring its data structure:

```
int64 num
```

2. Create custom srv definitions

Back in the `my_interfaces/srv` directory you just created, make a new file called `AddThreeInts.srv` with the following request and response structure:

```
int64 a  
int64 b  
int64 c  
---  
int64 sum
```

3. Create custom action definitions

Back in the `my_interfaces/action` directory you just created, make a new file called `Fibonacci.action` with the following request and response structure:

```
#goal definition  
int32 order  
---
```

```
#result definition
int32[] sequence
---
#feedback
int32[] partial_sequence
```

4. Update CMakeLists.txt

To convert the interfaces you defined into language-specific code (like C++ and Python) so that they can be used in those languages, add the following lines:

```
find_package(rosidl_default_generators REQUIRED)

rosidl_generate_interfaces(${PROJECT_NAME}
  # Insert all the messages, services and actions here
  "msg/Num.msg"
  "srv/AddThreeInts.srv"
  "action/Fibonacci.action"
)
```

5. Update package.xml

Because the interfaces rely on `rosidl_default_generators` for generating language-specific code, you need to declare a dependency on it. Add the following lines before the `</package>` tag:

```
<!-- _____Don't modify_____ -->
<depend>geometry_msgs</depend>
<depend>action_msgs</depend>
<buildtool_depend>rosidl_default_generators</buildtool_depend>
<exec_depend>rosidl_default_runtime</exec_depend>
<member_of_group>rosidl_interface_packages</member_of_group>
<!-- _____ -->
</package>
```

6. Build the workspace

After you have created your custom interfaces, you need to build the workspace:

```
colcon build --symlink-install
```

7. Confirm msg and srv creation

You can confirm that the messages and services were created using the following commands:

```
ros2 interface show my_interfaces/msg/Num  
ros2 interface show my_interfaces/srv/AddThreeInts  
ros2 interface show my_interfaces/action/Fibonacci
```


Create a Publisher and Subscriber with my_interfaces

In this section we will create the same publisher and subscriber as before, but this time we will use the custom messages created in `my_interfaces` package.

C++ Publisher and Subscriber with my_interfaces

1. Create a C++ package

Use the following command to create a new C++ package:

```
ros2 pkg create --build-type ament_cmake --license Apache-2.0 --node-name
<node_name>_node <package_name>_pkg
```

2. Create a C++ publisher and Subscriber

Go into `template_ws` in which you can find a `robot_v2_pkg` package. His node, `robot_v2_node`, is a simple publisher and subscriber. You can use it as a template to create your own publisher and subscriber.

3. Add dependencies

You need to add the dependencies of your package in the `package.xml` and in the `CMakeLists.txt` files. Go into `template_ws` in which you can find a `robot_v2_pkg` package. His `package.xml` and `CMakeLists.txt` files are a good example to add the dependencies.

Python Publisher and Subscriber with my_interfaces

1. Create a Python package

Use the following command to create a new Python package:

```
ros2 pkg create --build-type ament_python --license Apache-2.0 --node-name
<node_name>_node <package_name>_pkg
```

2. Create a Python publisher and Subscriber

Go into `template_ws` in which you can find a `camera_v2_pkg` package. His node, `camera_v2_node`, is a simple publisher and subscriber. You can use it as a template to create your own publisher and subscriber.

3. Add dependencies

You need to add the dependencies of your package in the `package.xml` and in the `setup.py` files. Go into `template_ws` in which you can find a `camera_v2_pkg` package. His `package.xml` and `setup.py` files are a good example to add the dependencies.

Create Service and Client using my_interface

C++ Service and Client with my_interfaces

1. Create a C++ package

Use the following command to create a new C++ package:

```
ros2 pkg create --build-type ament_cmake --license Apache-2.0 --node-name  
<node_name>_node <package_name>_pkg
```

2. Create a C++ Service Server and Client

Go into `template_ws` in which you can find a `robot_srv_pkg` package. His nodes are a simple service and client. You can use it as a template to create your own service and client.

3. Add dependencies

You need to add the dependencies of your package in the `package.xml` and in the `CMakeLists.txt` files. Go into `template_ws` in which you can find a `robot_srv_pkg` package. His `package.xml` and `CMakeLists.txt` files are a good example to add the dependencies.

Python Service and Client with my_interfaces

1. Create a Python package

Use the following command to create a new Python package:

```
ros2 pkg create --build-type ament_python --license Apache-2.0 --node-name  
<node_name>_node <package_name>_pkg
```

2. Create a Python Service Server and Client

Go into `template_ws` in which you can find a `camera_srv_pkg` package. His nodes are a simple service and client. You can use it as a template to create your own service and client.

3. Add dependencies

You need to add the dependencies of your package in the `package.xml` and in the `setup.py` files. Go into `template_ws` in which you can find a `camera_srv_pkg` package. His `package.xml` and `setup.py` files are a good example to add the dependencies.

Create Action Server and Client using my_interface

C++ Action Server and Client with my_interfaces

1. Create a C++ package

Use the following command to create a new C++ package:

```
ros2 pkg create --build-type ament_cmake --license Apache-2.0 --node-name  
<node_name>_node <package_name>_pkg
```

2. Create a C++ Action Server and Client

Go into `template_ws` in which you can find a `robot_action_pkg` package. His nodes are a simple service and client. You can use it as a template to create your own service and client.

3. Add dependencies

You need to add the dependencies of your package in the `package.xml` and in the `CMakeLists.txt` files. Go into `template_ws` in which you can find a `robot_srv_pkg` package. His `package.xml` and `CMakeLists.txt` files are a good example to add the dependencies.

Python Action Server and Client with my_interfaces

1. Create a Python package

Use the following command to create a new Python package:

```
ros2 pkg create --build-type ament_python --license Apache-2.0 --node-name  
<node_name>_node <package_name>_pkg
```

2. Create a Python Service Server and Client

Go into `template_ws` in which you can find a `camera_action_pkg` package. His nodes are a simple service and client. You can use it as a template to create your own service and client.

3. Add dependencies

You need to add the dependencies of your package in the `package.xml` and in the `setup.py` files. Go into `template_ws` in which you can find a `camera_srv_pkg` package. His `package.xml` and `setup.py` files are a good example to add the dependencies.