ROS2 Basics in 5 Days (Python)

Unit 2 Basic Concepts

- Summary -

Estimated time to completion: **1.5 hours**

What will you learn in this unit?

- How to structure and launch ROS2 programs (packages and launch files)
- How to create basic ROS2 programs (Python-based)
- Basic ROS2 concepts: nodes, client libraries, etc.

- End of Summary -

2.1 What is ROS2?

It is possible you took this course to answer this question: **"What is ROS2?"**

For now, however, it will be more helpful to experience what ROS2 can do.

2.1.1 Move a Robot with ROS2

On the right corner of the screen, you have your first simulated robot: the TurtleBot3. **Let us move that robot now!**

How do you move the TurtleBot?

The simplest method is to control the robot using an existing ROS2 program. Then, the executables created during the compilation of a ROS2 program are used to run it. Later in this guide, you will learn more about compilation.

Since it already exists in this workspace, you will launch a previously-made ROS2 program (executable) that allows you to move the robot using the keyboard.

- Example 2.1 -

Before attempting to run the executable, you will need to do some preliminary work. Do not think about the meanings of the commands below; you will learn how to use them during this tutorial.

So, get on with it. To source your working space, execute the following commands in **Webshell #1**:

Execute in Shell #1

In [ ]:

```
source /opt/ros/humble/setup.bash
```

In [ ]:

```
source /home/simulations/ros2_sims_ws/install/setup.bash
```

In [ ]:

```
ros2 run turtlebot3_teleop teleop_keyboard
```

CONGRATULATIONS! You launched your first ROS2 program! In this second terminal, you should have received a message similar to the following:

Shell #1 Output

```
Control Your TurtleBot3!
---------------------------
Moving around:
        w
   a    s    d
        x

w/x : increase/decrease linear velocity (Burger : ~ 0.22, Waffle and Waffle Pi : ~ 0.26)
a/d : increase/decrease angular velocity (Burger : ~ 2.84, Waffle and Waffle Pi : ~ 1.82)

space key, s : force stop

CTRL-C to quit
```

Now, you can use the keys indicated in the Webshell Output to move the robot around.

Okay. Try to move the robot around now!

REMEMBER, you need to focus on the terminal (Shell) where you launched the program for the keys to take effect. You know you have correctly focused when the cursor starts blinking.

When you get tired of playing with the robot, you can press **Ctrl+C** to stop the program's execution (remember to have the focus). You can close Webshell #1 to continue the course if you want.

Take a look back at what you have learned so far. The `ros2` keyword is used for all the ROS2 commands. Therefore, for launching programs, you will have two options:

- Launch the ROS2 program by directly running the **executable file**.
- Launch the ROS2 program by starting a **launch file**.

It may seem easier to use the run command to launch executables, but you will later understand why the launch command is also useful.

For now, you can directly run the executable file. The structure of the command is as follows:

In [ ]:

```
ros2 run <package_name> <executable_file>
```

As you can see, the command has two parameters: the first parameter is **the name of the package** that contains the executable file. The second parameter is **the name of the executable file** (which is stored inside the package).

For using a launch file, the structure of the command would go as follows:

In [ ]:

```
ros2 launch <package_name> <launch_file>
```

As you can see, this command also has two parameters: the first parameter is **the name of the package** that contains the launch file. The second parameter is **the name of the launch file** (which is stored in the package).

- End of Example 2.1 -

2.2 What is a Package?

ROS2 uses **packages** to organize its programs. You can think of a package as **all the files that a specific ROS2 program contains**; all its CPP files, Python files, configuration files, compilation files, launch files, and parameters files. Also, organizing your ROS2 programs in packages makes sharing them with other developers/users much easier.

In ROS2, you can create two types of packages: Python packages and CMake (C++) packages. For this course, though, we will focus on the first ones. Python packages will contain Python executables.

Every Python package will have the following structure of files and folders:

- `package.xml` - File containing meta-information about the package (maintainer of the package, dependencies, etc.).
- `setup.py` - File containing instructions for how to compile the package.
- `setup.cfg` - File that defines where the scripts will be installed.
- `/<package_name>` - This directory will always have the same name as your package. You will put all your Python scripts inside this folder. Note that it already contains an empty `__init__.py` file.

Some packages might contain extra folders. For instance, the `launch` folder contains the package's launch files (you will read more about it later).

To summarize, you should remember the following:

- Every ROS2 program that you want to execute is organized in a package.
- Every ROS2 program you create must be organized in a package.
- Packages are the primary organization system for ROS2 programs.

2.3 Create a Package

Until now, you have been inspecting the layout of an existing build package. Now, it is time to make your own.

When you want to create packages, you need to work in a particular workspace known as the **ROS2 workspace**. The ROS2 workspace is the directory in your hard disk where your **ROS2 packages reside** to be usable by ROS2. Usually, the **ROS2 workspace** directory is called **ros2_ws**.

- Example 2.2 -

First, source ROS2 in your shell to use the ROS2 command-line tools.

Execute in Shell #1

In [ ]:

```
source /opt/ros/humble/setup.bash
```

Now, go to the **ros2_ws** in your Webshell #1

In [ ]:

```
cd ~/ros2_ws/
```

In [ ]:

```
pwd
```

This will give you the following as output:

Shell #1 Output

```
/home/user/ros2_ws
```

Inside this workspace, there is a directory called **src**. This folder contains all the packages created. Every time you want to create a package, you have to be in this directory **ros2_ws/src**. Type into your Webshell the following command:

Execute in Shell #1

In [ ]:

```
cd src
```

At this point, you are finally ready to create your own package! To do that, type the following into your Webshell:

In [ ]:

```
ros2 pkg create --build-type ament_python my_package --dependencies rclpy
```

Something similar to the message below will appear in your terminal:

Shell #1 Output

```
going to create a new package
package name: my_package
destination directory: /home/user/ros2_ws
package format: 3
version: 0.0.0
description: TODO: Package description
maintainer: ['user ']
licenses: ['TODO: License declaration']
build type: ament_python
dependencies: ['rclpy']
creating folder ./my_package
creating ./my_package/package.xml
creating source folder
creating folder ./my_package/my_package
creating ./my_package/setup.py
creating ./my_package/setup.cfg
creating folder ./my_package/resource
creating ./my_package/resource/my_package
creating ./my_package/my_package/__init__.py
creating folder ./my_package/test
creating ./my_package/test/test_copyright.py
creating ./my_package/test/test_flake8.py
creating ./my_package/test/test_pep257.py
creating ./my_package/my_package/my_node.py
```

Inside your **src** directory, this creates a new package with files. You will check this later. Now, see how this command is built:

In [ ]:

```
ros2 pkg create --build-type ament_python <package_name> --dependencies <package_dependencies>
```

The `<package_name>` is the name of the package you want to create, and the `<package_dependencies>` are the names of other ROS2 packages that your package depends on.

Note also that we are specifying `ament_python` as the `build type`. This indicates that we are creating a Python package.

It is a good idea to build your package after it has been created. It is the quickest way to determine if the dependencies you listed can be resolved and check that there are no mistakes in the entered data.

In [ ]:

```
cd ~/ros2_ws/
```

In [ ]:

```
colcon build
```

Make it a habit to source setup.bash from the install folder so that ROS can find the packages in the workspace.

In [ ]:

```
source install/setup.bash
```

- End of Example 2.2 -

- Example 2.3 -

To confirm that your package has been created successfully, use some ROS commands related to packages. For example, type the following:

Execute in Shell #1

In [ ]:

```
ros2 pkg list
```

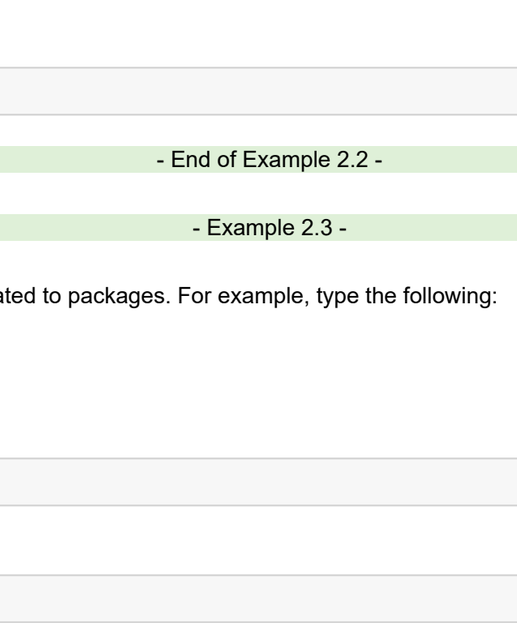
In [ ]:

```
ros2 pkg list | grep my_package
```

ros2 pkg list: Gives you a list of all your ROS system packages.

ros2 pkg list | grep my_package: Filters, from all of the packages located in the ROS system, the package is named my_package.

You can also see the package created and its contents by opening it through the IDE:



If the above commands have not shown you anything in the terminal, do not panic. It is normal. If you have worked with code before, you should remember that you must first compile your code to get executables, so let us get on with it in the next section.

- End of Example 2.3 -

- Notes -

Packages are organized inside workspaces. Each workspace can contain as many packages as you want. For this course, your workspace is named `ros2_ws`. So, the overall structure would look as follows:

```
ros2_ws/
    src/
        my_package/
            package.xml
            setup.py
            ...
        my_package_2/
            package.xml
            setup.py
            ...
        my_package_x/
            package.xml
            setup.py
            ...
```

- End of Notes -

2.4 Compile a Package

When you create a package, you need to compile it to make it work. The command used by ROS2 to compile is the following:

In [ ]:

```
colcon build
```

This command will compile your whole **src** directory, which needs to be issued in your **ros2_ws** directory to work. **Please, remember this!**

- Example 2.4 -

Go to your **ros2_ws** directory and compile your source folder. You can do so by typing the following:

Execute in Shell #1

In [ ]:

```
cd ~/ros2_ws
```

In [ ]:

```
colcon build
```

Also, after compiling, you have to **source** the workspace to make sure that the latest modifications/updates are taken into account:

Execute in Shell #1

In [ ]:

```
source install/setup.bash
```

Sometimes (for large projects), you will not want to compile all of your packages. This would take such a long time. So instead, you can use the following command to compile only the packages where you have made changes:

In [ ]:

```
colcon build --packages-select <package_name>
```

This command will only compile the packages specified and their dependencies.

Now, try to compile your package named **my_package** with this command.

Execute in Shell #1

In [ ]:

```
colcon build --packages-select my_package
```

- End of Example 2.4 -

2.5 What is a Launch File?

You have seen how ROS2 can run programs from launch files. However, how do they work? Take a look at it.

- Example 2.5 -

In **Example 2.1**, you used the command ROS2 run to start the **teleop_keyboard** executable file. However, you have also seen that you can run executables using what you know as launch files.

How do they operate? Take a look at an example. If you want to use a launch file to start the **teleop_keyboard** executable, you would need to write something similar to the Python script below:

In [ ]:

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

def generate_launch_description():
    return LaunchDescription([
        launch_ros.actions.Node(
            package='turtlebot3_teleop',
            executable='teleop_keyboard',
            output='screen'),
    ])
```

Do not worry! You do not need to write these lines of code in your workspace. Instead, you will use them to review some concepts.

Okay. As you can see, the launch file structure is quite simple. First, you import some modules from the launch and launch_ros packages.

In [ ]:

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

Next, define a function that will return a LaunchDescription object.

In [ ]:

```
def generate_launch_description():
    return LaunchDescription([
        launch_ros.actions.Node(
            package='teleop_twist_keyboard',
            executable='teleop_twist_keyboard',
            output='screen'),
    ])
```

Within the LaunchDescription object, generate a node where you will provide the following parameters:

1. `package='package_name'` Name of the package that contains the code of the ROS2 program to execute
2. `executable='python_executable_name'` Name of the Python executable file that you want to execute
3. `output='type_of_output'` Through which channel you will print the output of the program

- End of Example 2.5 -

- Notes -

Launch files are useful since they provide an easy way to start multiple nodes with a single file, as well as the possibility to configure these nodes by, for instance, setting parameters. You can learn more details about launch files and work with parameters in ROS2 in our Intermediate ROS2 course.

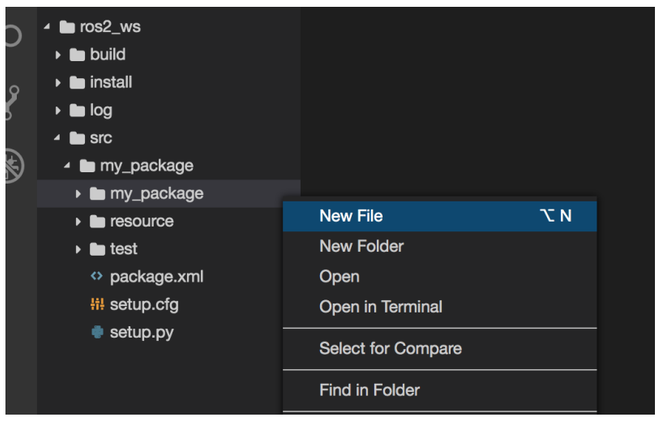
- End of Notes -

2.6 It Is Time for Your First ROS2 Program

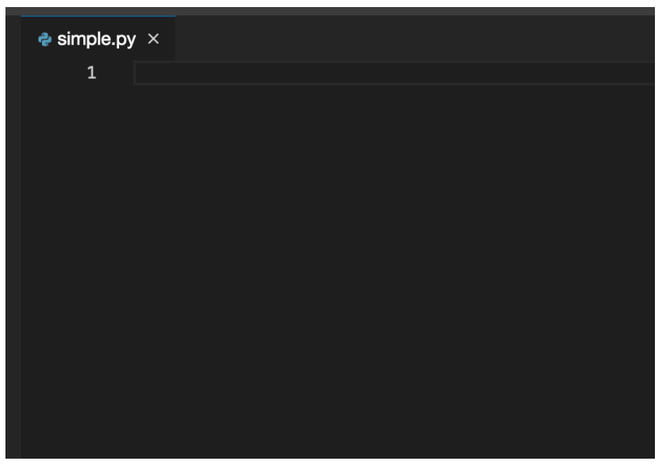
You should now have your first package built, and you must do something about it! So get started with your first ROS2 program!

- Example 2.6 -

1. Create a Python file in the **my_package** directory, which is inside the **my_package** package (I know, confusing, but trust me for a moment). For this exercise, copy this simple Python code **simple.py**. You can create it directly by RIGHT-clicking on the IDE in the **my_package** directory of your package and selecting **New File**.



A new Tab should have appeared on the IDE with empty content.

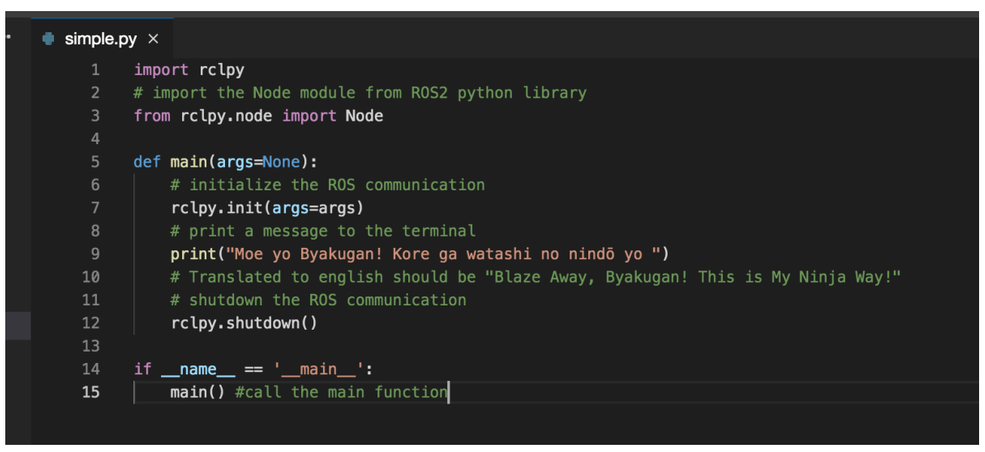


Now, copy the content of simple.py into the new file.

simple.py

In [ ]:

```
import rclpy
# import the Node module from ROS2 Python library
from rclpy.node import Node

def main(args=None):
    # initialize the ROS communication
    rclpy.init(args=args)
    # print a message to the terminal
    print("Moe yo Byakugan! Kore ga watashi no nindō yo ")
    # english translation: "Blaze Away, Byakugan! This is My Ninja Way!"
    # shutdown the ROS communication
    rclpy.shutdown()

if __name__ == '__main__':
    main() #call the main function
```



2. Create a launch directory inside the package **my_package**.

Execute in Shell #1

In [ ]:

```
cd ~/ros2_ws/src/my_package
```

In [ ]:

```
mkdir launch
```

You can also create the launch directory through the IDE, as you have seen before, but try practicing with the terminal.

3. Create a new launch file inside the launch directory.

Execute in Shell #1

In [ ]:

```
cd ~/ros2_ws/src/my_package/launch
```

In [ ]:

```
touch my_package_launch_file.launch.py
```

In [ ]:

```
chmod +x my_package_launch_file.launch.py
```

You can also create it through the IDE.

- Exercise 2.1 -

4. Fill the launch file using the knowledge you have acquired. Take a look at **Example 2.5**.

- End of Exercise 2.1 -

Were you able to do it on your own? If not, do not worry, and do not give up. The final launch should be something similar to the following: **my_package_launch_file.launch.py**.

- Solution for Exercise 2.1 -

In [ ]:

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

def generate_launch_description():
    return LaunchDescription([
        Node(
            package='my_package',
            executable='simple_node',
            output='screen'),
    ])
```

Wow! It was not too complicated.

- End of Solution for Exercise 2.1 -

5. Modify the setup.py file to generate an executable from your Python file.

- Notes -

I know this has been sudden, but do not worry. This is one of those moments when you have to trust me. Try to analyze the code in the cell below but do not worry if there is something you do not understand there. With practice throughout this course, you will master how to write a `setup.py` file.

- End of Notes -

Your final **`setup.py`** file should look like this:

setup.py

In [ ]:

```
from setuptools import setup
import os
from glob import glob

package_name = 'my_package'

setup(
    name=package_name,
    version='0.0.0',
    packages=[package_name],
    data_files=[
        ('share/ament_index/resource_index/packages',
            ['resource/' + package_name]),
        ('share/' + package_name, ['package.xml']),
        (os.path.join('share', package_name), glob('launch/*.launch.py'))
    ],
    install_requires=['setuptools'],
    zip_safe=True,
    maintainer='somebody very awesome',
    maintainer_email='user@user.com',
    description='TODO: Package description',
    license='TODO: License declaration',
    tests_require=['pytest'],
    entry_points={
        'console_scripts': [
            'simple_node = my_package.simple:main'
        ],
    },
)
```

6. Compile your package file as was previously explained.

Execute in Shell #1

In [ ]:

```
cd ~/ros2_ws
```

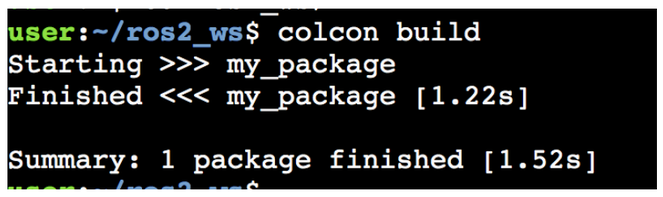
In [ ]:

```
colcon build
```

In [ ]:

```
source ~/ros2_ws/install/setup.bash
```

If everything goes as planned, you should get something like the following as output:



7. Finally, the time has come. Now you must execute it. Run the following command (that you already know) in the Webshell to launch your program.

Execute in Shell #1

In [ ]:

```
ros2 launch my_package my_package_launch_file.launch.py
```

Shell #1 Output

```
[INFO] [launch]: All log files can be found below /home/user/.ros/log/2021-04-08-21-08-07-034933-5_xterm-7778
[INFO] [launch]: Default logging verbosity is set to INFO
[INFO] [simple-1]: process started with pid [7780]
[simple-1] Moe yo Byakugan! Kore ga watashi no nindō yo
[INFO] [simple-1]: process has finished cleanly [pid 7780]
```

- End of Example 2.6 -

2.7 Modifying the setup.py File

The `setup.py` file contains all the necessary instructions for properly compiling your package.

In the previous exercise, you had the following file:

setup.py

In [ ]:

```
from setuptools import setup
import os
from glob import glob

package_name = 'my_package'

setup(
    name=package_name,
    version='0.0.0',
    packages=[package_name],
    data_files=[
        ('share/ament_index/resource_index/packages',
            ['resource/' + package_name]),
        ('share/' + package_name, ['package.xml']),
        (os.path.join('share', package_name), glob('launch/*.launch.py'))
    ],
    install_requires=['setuptools'],
    zip_safe=True,
    maintainer='somebody very awesome',
    maintainer_email='user@user.com',
    description='TODO: Package description',
    license='TODO: License declaration',
    tests_require=['pytest'],
    entry_points={
        'console_scripts': [
            'simple_node = my_package.simple:main'
        ],
    },
)
```

The main objective of this code is to generate an executable from the script you created a few moments ago. To do that, you work with a dictionary named `entry_points`. Inside it, you find an array called `console_scripts`.

Add the node information to generate the executable.

In [ ]:

```
import os
from glob import glob
from setuptools import setup

package_name = 'my_package'


setup(

    #code
    ...
    #code

    entry_points={
            'console_scripts': [
                'simple_node = my_package.simple:main'
            ],
        },

    #code
    ...

)
```

With these lines, you are adding an entry point to the script you wrote earlier, `simple.py`. For example, you can see this line as follows:

In [ ]:

```
'<executable_name> = <package_name>.<script_name>:main'
```

So, in this case, you are generating a new executable node named **`simple_node`**. This executable is generated using a script named **`simple.py`** inside a package named **`my_package`**.

Additionally, there are other things that you can define in a **`setup.py`** file.

For colcon to find the launch files during the compilation process, you need to inform Python's setup tools of your launch files using the **data_files** parameter of the **`setup.py`** file.

Have a look at the `data_files` array:

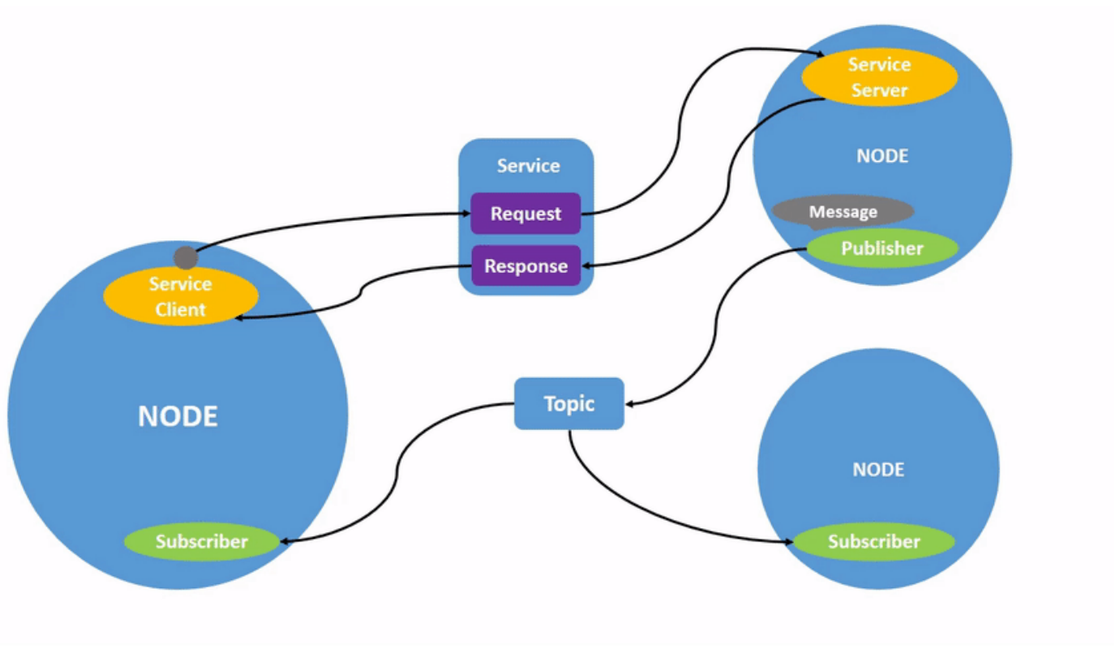
In [ ]:

```
import os
from glob import glob
from setuptools import setup

package_name = 'my_package'


setup(

    #code
    ...
    #code

    data_files=[
            ('share/ament_index/resource_index/packages',
                ['resource/' + package_name]),
            ('share/' + package_name, ['package.xml']),
            (os.path.join('share', package_name), glob('launch/*.launch.py'))
        ],

    #code
    ...
    #code

)
```

The objective of this code is to install the launch files. For example, with the package named **`my_package`**, this will install all the launch files from the `launch/` folder, into `~/ros2_ws/install/my_package/share/my_package/`.

2.8 ROS2 Nodes

In ROS2, each node should be responsible for a single module (e.g., one node for controlling wheel motors, one for controlling a laser range-finder, etc.). Each node can communicate with other nodes through different methods.



The animated image is taken from the official ROS2 documentation

For the moment, do not worry about understanding everything you see in the picture. Look at the nodes and how they connect. A full robotic system is comprised of many nodes working together. In ROS2, a single executable (a C++ or Python program, etc.) can contain one or more nodes.

Okay. See what you can do with these nodes!

In the previous code, you started a node. What is a node? ROS nodes are essentially ROS programs. To list all the nodes running on a system, use the ROS2 command:

In [ ]:

```
ros2 node list
```

Run the command:

Execute in Shell #1

In [ ]:

```
ros2 node list
```

Shell #1 Output

```
/camera_driver
/gazebo
/moving_service
/robot_state_publisher
/service
/turtlebot3_diff_drive
/turtlebot3_imu
/turtlebot3_joint_state
/turtlebot3_laserscan
```

The nodes you see right now are all related to the Gazebo simulation. Follow the next example to run and visualize your own node.

- Example 2.7 -

First, update your Python file `simple.py` with the following code:

In [ ]:

```
import rclpy
from rclpy.node import Node

class MyNode(Node):
    def __init__(self):
        # call super() in the constructor to initialize the Node object
        # the parameter we pass is the node name
        super().__init__('Byakugan')
        # create a timer sending two parameters:
        # - the duration between two callbacks (0.2 seconds)
        # - the timer function (timer_callback)
        self.create_timer(0.2, self.timer_callback)

    def timer_callback(self):
        # print a ROS2 log on the terminal with a great message!
        self.get_logger().info("Moe yo Byakugan! Kore ga watashi no nindō yo")

def main(args=None):
    # initialize the ROS2 communication
    rclpy.init(args=args)
    # declare the node constructor
    node = MyNode()
    # keeps the node alive, waits for a request to kill the node (ctrl+c)
    rclpy.spin(node)
    # shutdown the ROS2 communication
    rclpy.shutdown()

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

Have a quick look at the new code. First, create a `MyNode` class, which inherits from `Node`.

In [ ]:

```
class MyNode(Node):
```

The class has two methods: `__init__` (the constructor of the class) and `timer_callback`.

Inside the `__init__` method, you are initializing a node named **Byakugan**:

In [ ]:

```
super().__init__('Byakugan')
```

Also, create a timer object:

In [ ]:

```
self.create_timer(0.2, self.timer_callback)
```

This timer object will trigger the `timer_callback` method every 0.2 seconds.

- Notes -

Timer objects are useful in ROS2. You will get more familiar with them as you use them during the course.

- End of Notes -

Inside the `timer_callback` method, you sent a message to the node's log.

In [ ]:

```
self.get_logger().info("Moe yo Byakugan! Kore ga watashi no nindō yo")
```

- Notes -

You will learn about the ROS2 log system in **Unit 6 - Debugging Tools**.

- End of Notes -

Finally, in your main function, create an instance of the class `MyNode` and spin it.

In [ ]:

```
node = MyNode()

rclpy.spin(node)
```

As you can see in the code comments, `spin()` will keep the node alive and running until someone shuts it down (by pressing `Ctrl+C`).

The node will keep publishing the log messages until you stop it.

Recompile it once you have made some changes to your code.

Execute in Shell #1

In [ ]:

```
cd ~/ros2_ws
```

In [ ]:

```
colcon build
```

In [ ]:

```
source ~/ros2_ws/install/setup.bash
```

Now launch the program.

Execute in Shell #1

In [ ]:

```
ros2 launch my_package my_package_launch_file.launch.py
```

Try again in another terminal.

Execute in Shell #2

In [ ]:

```
source /opt/ros/humble/setup.bash
```

In [ ]:

```
ros2 node list
```

Shell #2 Output

```
/Byakugan
/camera_driver
/gazebo
/moving_service
/robot_state_publisher
/service
/turtlebot3_diff_drive
/turtlebot3_imu
/turtlebot3_joint_state
/turtlebot3_laserscan
```

As you can see, now the `/Byakugan` node appears in the list!

To see information about your node, you can use the following command:

In [ ]:

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

This command shows you information about all the connections that your node has.

Execute in Shell #2

In [ ]:

```
ros2 node info /Byakugan
```

Shell #2 Output

```
/Byakugan
  Subscribers:

  Publishers:
    /parameter_events: rcl_interfaces/msg/ParameterEvent
    /rosout: rcl_interfaces/msg/Log
  Service Servers:
    /Byakugan/describe_parameters: rcl_interfaces/srv/DescribeParameters
    /Byakugan/get_parameter_types: rcl_interfaces/srv/GetParameterTypes
    /Byakugan/get_parameters: rcl_interfaces/srv/GetParameters
    /Byakugan/list_parameters: rcl_interfaces/srv/ListParameters
    /Byakugan/set_parameters: rcl_interfaces/srv/SetParameters
    /Byakugan/set_parameters_atomically: rcl_interfaces/srv/SetParametersAtomically
  Service Clients:

  Action Servers:

  Action Clients:
```

Do not be concerned about the whole command's output for now. You will gain a better understanding as you progress through the next lessons.

- End of Example 2.7 -

2.9 Client Libraries

You used the rclpy client library in the previous exercise. What is this? In a nutshell, ROS client libraries allow nodes written in various programming languages to communicate. A core ROS client library (RCL) implements the standard functionality needed by various ROS APIs. This makes it easier to write language-specific client libraries.

The ROS2 team currently maintains the following client libraries:

- rclcpp = C++ client library
- rclpy = Python client library

Additionally, other client libraries have been developed by the ROS community. You can check out the following article for more details: here.

2.10 We Are Here! So, What is ROS2?

ROS2 is the platform that helps you do everything you have seen so far in this unit. It provides the foundation for managing all of these processes and their interactions. You have just scratched the surface of ROS2's fundamental principles in this tutorial.

ROS2 is a powerful tool, sometimes difficult, but powerful. So, if you take your lessons and continue on your ninja way to learn ROS2, you will do wonderful things with fantastic robots at some point.

English proofread