Part 2 Developing with ROS 2 Python and C++

Open a terminal, navigate to your home directory, and create the workspace:

```
$ cd
$ mkdir ros2_ws
```

Then, enter the workspace and create a new directory named src. This is where you will write all
the code for your ROS 2 application:

```
$ cd ros2_ws/
$ mkdir src
```

To set up a new workspace, you just create a new directory (somewhere in your home directory) and create an **src** directory inside it.

Building the workspace
1.Navigate to the workspace root directory. Make sure you are in the right place.
2.Run the colcon build command. colcon is the build system in ROS 2, and it was installed when you installed the ros-dev-tools packages in Chapter 2.

```
$ cd ~/ros2_ws/
$ colcon build
Summary: 0 packages finished [0.73s]
```

As you can see, no packages were built, but let's list all directories under ~/ros2_ws:

```
$ ls
build install log src
```

The build directory will contain the intermediate files required for the overall build.
In log, you will find logs for each build.
The install is where all your nodes will be installed after you build the workspace.

`Note: You should always run colcon build from the root of your workspace directory, not from anywhere else. If you make a mistake and run this command from another directory (let's say, from the src directory of the workspace, or inside a package), simply remove the new install, build, and log directories` `that were created in the wrong place. Then go back to the workspace root directory and build again.`

Sourcing the workspace

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

```
$ cd install/
$ ls
COLCON_IGNORE    _local_setup_util_ps1.py  setup.ps1
local_setup.bash  _local_setup_util_sh.py  setup.sh
local_setup.ps1   local_setup.zsh      setup.zsh
local_setup.sh   setup.bash
```

Every time you build your workspace, you have to source it so that the environment (the session you are in) knows about the new changes in the workspace.
To source the workspace, source this setup.bash script:

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

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

Open your .bashrc (located in your home directory the path is ~/.bashrc) using any text editor you want:

```
$ gedit ~/.bashrc
```

Add the line to source the workspace's setup.bash script, just after the one to source the global
ROS 2 installation. The order is very important here. You have to source the global ROS 2 installation first, and then your workspace, not the other way around:

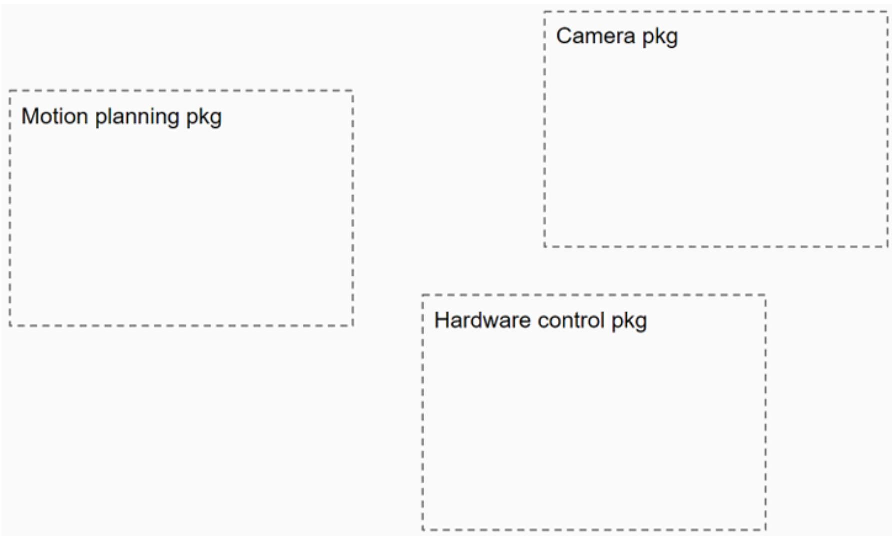
```
source /opt/ros/jazzy/setup.bash
source ~/ros2_ws/install/setup.bash
```

Make sure to save.bashrc. Now, both ROS 2 and your workspace will be sourced in any new terminal you open.

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

Creating a package
Any node you create will exist within a package. Hence, to create a node, you first have to create a
package (inside your workspace)

What is a ROS 2 package?
A ROS 2 package is a sub-part of your application.
Let's consider a robotic arm that we want to use to pick up and place objects. Before creating any
node, we can try to split this application into several sub-parts, or packages: one package to handle a camera, another package for the hardware control (motors), and yet another package to compute motion planning for the robot.



Each package is an independent unit, responsible for one sub-part of your application. Packages are very useful for organizing your nodes, and also to correctly handle dependencies

Creating a Python package
You will create all your packages in the src directory of your ROS 2 workspace. So, make sure to
navigate to this directory before you do anything else:

```
$ cd ~/ros2_ws/src/
```

Here is how to construct the command to create a package:

1. ros2 pkg create <pkg_name>: This is the minimum you need to write.
2. You can specify a build type with --build_type <build_type>. For a Python package, we need to use ament_python.

3. You can also specify some optional dependencies with --dependencies <list*of* dependencies_separated_with_spaces>. It’s always possible to add dependencies later in the package.

In the src directory of your workspace, run the following:

```
$ ros2 pkg create my_py_pkg --build-type ament_python --dependencies
rclpy
```

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

This will print quite a few logs, showing you what files have been created. You might also get a [WARNING] log about a missing license, but as we have no intention of publishing this package
anywhere, we don’t need a license file now. You can ignore this warning.

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

```
/home/<user>/ros2_ws/src/my_py_pkg
├── my_py_pkg
|  └── __init__.py
├── package.xml
├── resource
|  └── my_py_pkg
├── setup.cfg
├── setup.py
└── test
   ├── test_copyright.py
   ├── test_flake8.py
   └── test_pep257.py
```

Quick Overview
my_py_pkg: As you can see, inside the package, there is another directory with the same name. This directory already contains an **init**.py file. This is where we will create our Python nodes.
package.xml: Every ROS 2 package (Python or C++) must contain this file. We will use it to provide more information about the package as well as dependencies.
setup.py: This is where you will write the instructions to build and install your Python nodes.

Building a package
Now that you've created one or more packages, you can build them, even if you don't have any nodes
in the packages yet.
To build the packages, go back to the root of your ROS 2 workspace and run colcon build. Once
again, and as seen previously in this chapter, where you run this command is very important.

```
~/ros2_ws/src$ cd ~/ros2_ws/
~/ros2_ws$ colcon build
Starting >>> my_py_pkg
Finished <<< my_py_pkg [2.56s]
Summary: 1 package finished [2.83s]
```

Both packages have been built. You will have to do that every time you add or modify a node inside
a package.

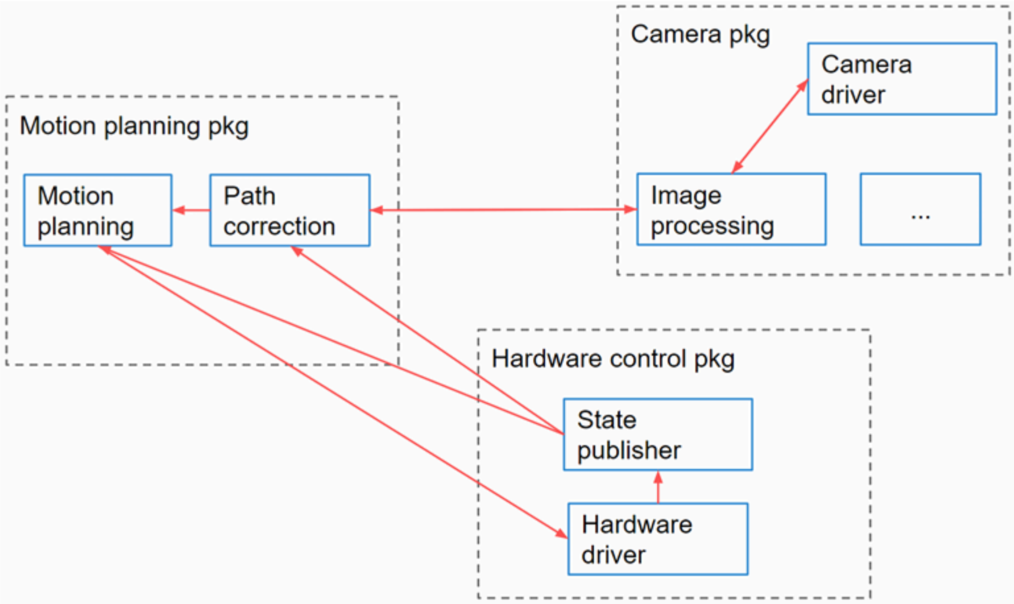
The important thing to notice is this line: Finished <<< <package_name> [time]. This means that the package was correctly built. Even if you see additional warning logs, if you also see the Finished line, you know the package has been built.

To build only a specific package, you can use the --packages-select option, followed by the name of the package:

```
$ colcon build --packages-select my_py_pkg
Starting >>> my_py_pkg
Finished <<< my_py_pkg [1.01s]
Summary: 1 package finished [1.26s]
```

How are nodes organized in a package?

To develop a ROS 2 application, you will write code inside nodes. A node is a subprogram of your application, responsible for one thing. If you have two different functionalities to implement, then you will have two nodes. Nodes communicate with each other using ROS 2 communications (topics, services, and actions).
You will organize your nodes inside packages. For one package (sub-part of your application), you
can have several nodes (functionalities).



The **camera package** includes a node that handles the camera and sends images to an **image processing node**, which extracts object coordinates.
The **motion planning package** contains a node that calculates robot movements based on commands, assisted by a **path correction node** using image data.
A **hardware driver node** executes these movements by communicating with motors and encoders. A **state publisher node** shares robot status data with other nodes.

Creating a file for the node

For every Python package, you have to go to the directory which has the same name as the package. If your package name is abc, then you'll go to ~/ros2_ws/src/abc/abc/.
Create a new file in this directory and make it executable:

```
$ cd ~/ros2_ws/src/my_py_pkg/my_py_pkg/
$ touch my_first_node.py
$ chmod +x my_first_node.py
```

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

```
$ cd ~/ros2_ws/src/
$ code .
```

1. Shebang Line

```
#!/usr/bin/env python3
```

- This tells the system to use **Python 3** to run the script.

- It's helpful when running the script as an executable (like `./script.py`).

2. Import ROS 2 Python Library

```
import rclpy
from rclpy.node import Node
```

- `rclpy` is the Python client library for ROS 2.
- From it, we import the `Node` class, which is the base class for creating nodes in ROS 2.

3. Define a Custom Node Class

```
class MyCustomNode(Node):
    def __init__(self):
        super().__init__('my_node_name')
        self.get_logger().info("Hello World")
```

- `MyCustomNode` is your custom ROS 2 node.
- It **inherits from** `Node`, so it gets all the functionality a ROS 2 node needs.
- `super().__init__('my_node_name')` initializes the parent `Node` class with a name.
- `self.get_logger().info("Hello World")` logs a message when the node is created.

4. Define the `main()` Function

```
def main(args=None):
    rclpy.init(args=args)
    node = MyCustomNode()
    rclpy.spin(node)
    rclpy.shutdown()
```

This function handles the node's lifecycle:

1. **Initialize ROS 2**:

   ```
   rclpy.init(args=args)
   ```

   - Sets up ROS 2 communication.
2. **Create Node Instance**:

   ```
   node = MyCustomNode()
   ```

   - Instantiates your custom node class.
3. **Spin the Node**:

```
rclpy.spin(node)
```

- Keeps the node active and responsive to callbacks.
- Blocks the script here so it doesn't exit immediately.
- Ends when you press **Ctrl+C**.

4. **Shutdown ROS 2**:

```
rclpy.shutdown()
```

- Cleans up all ROS 2 resources before the program exits.

5. Entry Point

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

- This is **standard Python**.
- Ensures that `main()` runs only if you execute the script directly (not when imported as a module).

Key Concepts

- A **node** is just a Python object.
- **Spinning** = keeping the node alive and processing incoming events.
- ROS 2 **initialization and shutdown** must be explicitly handled in Python.

Building the node

The word "build" is used, because to install a Python node, we have to run colcon build.
Open the setup.py file from
the my_py_pkg package. Locate entry_points and 'console_scripts' at the end of the
file. For each node we want to build, we have to add one line inside the 'console_scripts' array:

```
entry_points={
  'console_scripts': [
    "test_node = my_py_pkg.my_first_node:main"
  ],
},
```

syntax:

```
<executable_name> = <package_name>.<file_name>:<function_name>.
```

There are a few important things to correctly write this line:

- First, choose an executable name. This will be the name you use with ros2 run <pkg_name><executable_name>.
- For the filename, skip the .py extension.
- The function name is main, as we have created a main() function in the code.
- If you want to add another executable for another node, don't forget to add a comma between each executable and place one executable per line.

```
Note: When learning ROS 2, there is a common confusion between the node
name, filename, and executable name:
Node name: defined inside the code, in the constructor. This is what you'll
see with the ros2
node list, or in rqt_graph.
- Filename: the file where you write the code.
- Executable name: defined in setup.py and used with ros2 run.
In this first example, I made sure to use a different name for each so you
can be aware that
these are three different things. But sometimes all three names could be the
same. For example,
you could create a temperature_sensor.py file, then name your node and your
executable temperature_sensor.
```

Go to your workspace root directory and build the package:

```
$ cd ~/ros2_ws/
$ colcon build
```

You can also add --packages-select my_py_pkg to only build this package.
The executable should now be created and installed in the workspace (it will be placed inside the install directory).

Running the node

Now you can run your first node, but just before that, make sure that the workspace is sourced in your environment:

`$ source ~/.bashrc`

This file already contains the line to source the workspace; you could also just open a new terminal, or source the setup.bash script from the workspace.
Now run your node using ros2 run

```
$ ros2 run my_py_pkg test_node
[INFO] [1710922181.325254037] [my_node_name]: Hello World
```

Improving the node – timer and callback

Lets make the node print a string every second, as long as it's alive.
This behavior of "doing X action every Y seconds" is very common in robotics. For example, one
could have a node that "reads a temperature every 2 seconds", or that "gives a new motor command every 0.1 seconds".
We will add a timer to our node. A timer will trigger a callback function at a specified rate.

modify the MyCustomNode class. The rest of the code stays the same:

```python
class MyCustomNode(Node):
  def __init__(self):
    super().__init__('my_node_name')
    self.counter_ = 0
    self.timer_ = self.create_timer(1.0, self.print_hello)
  def print_hello(self):
    self.get_logger().info("Hello " + str(self.counter_))
    self.counter_ += 1
```

The constructor still uses `super()`, but now the logging is moved to a separate method. Instead of just printing "Hello World," a `counter_` attribute is added and incremented with each log.

The trailing underscore (_) in attribute names is a **coding convention** to indicate class attributes—purely for clarity and consistency.

A **timer** is created using `create_timer(rate, callback)`, where `rate` is the interval (e.g., `1.0` seconds) and `callback` is the method to call (without parentheses). In this case, `print_hello` is called every second.

To run the updated node:

- Always follow the cycle: **build → source → run**
- Do this every time you create or modify a node.

Use the `--symlink-install` option with `colcon build` to avoid rebuilding your Python package every time you change the code:

```
colcon build --packages-select my_py_pkg --symlink-install
```

- This works **only for Python packages**.

- You may see warnings, but if you see `Finished <<< my_py_pkg`, it built successfully.
- You **still need to build once** for any **new executable** you add.

The `print_hello()` method isn't called directly—it's registered as a **callback** using `create_timer()`. This works because the node is **spinning** via `rclpy.spin(node)`, which keeps the node alive and allows callbacks to execute.

This is your first example of a **callback** in ROS 2. Nearly everything in ROS 2 relies on callbacks. If the syntax or flow isn't clear yet, don't worry—it becomes clearer with hands-on practice and repetition as you progress.

Template for a Python node

```
#!/usr/bin/env python3
import rclpy
from rclpy.node import Node
class MyCustomNode(Node): # MODIFY NAME
  def __init__(self):
    super().__init__("node_name") # MODIFY NAME
def main(args=None):
rclpy.init(args=args)
  node = MyCustomNode() # MODIFY NAME
  rclpy.spin(node)
  rclpy.shutdown()
if __name__ == "__main__":
  main()
```

Remove the MODIFY NAME comments and change the class name (MyCustomNode) and the node name ("node_name"). It's better to use names that make sense. For example, if you are writing a node to read data from a temperature sensor, you could name the class TemperatureSensorNode, and the node could be temperature_sensor.

Introspecting your nodes

Being able to **inspect your nodes** helps you debug your code and understand other nodes you didn't write. The command-line tool for this is:

```
ros2 node
```

Before using it, make sure a node is running. To start a node, use:

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

For example, to run the Python node created in this chapter:

```
ros2 run my_py_pkg my_node_exe
```

Once running, you can explore its details using the `ros2 node` commands.

Changing the node name at run time

You can **rename a node at runtime** when using `ros2 run`. To pass additional arguments, use:

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

- `--ros-args` starts the section for ROS-specific arguments.
- `-r __node:=abc` or `--remap __node:=abc` changes the node's name to `abc`.

This is useful when you want to run multiple instances of the same node or avoid name conflicts.

```
Note: When running multiple nodes, you should make sure that each node has a
unique name. Having two nodes with the same name can lead to some unexpected
issues that can take a long time to debug. In the future, you will see that
you may want to run the same node several times, for example, three
temperature_sensor nodes, one each for a different sensor. You could
rename them so that you have temperature_sensor_1, temperature_sensor_2,
and temperature_sensor_3.
```