First Node

Polytech Angers - Mobile Robotics

This part is widely inspired from https://docs.ros.org/en/humble/Tutorials/Beginner-Client-Libraries/Colcon-Tutorial.html

Setting up a workspace

> A ROS workspace is a directory with a particular structure. Commonly there is a src subdirectory. Inside that subdirectory is where the source code of ROS packages will be located. Typically, the directory starts otherwise empty.
> https://docs.ros.org/en/humble/Tutorials/Beginner-Client-Libraries/Colcon-Tutorial.html

Create a `src` directory into your working directory (should be `wdir`):

```
docker@ros2:~/wdir$ mkdir src
```

From there, we will use the `colcon` utility to create our ROS 2 packages and executables. First you can check that everything is set up properly and create the workspace architecture:

```
docker@ros2:~/wdir$ colcon build

Summary: 0 packages finished [0.18s]
```

Now in your working directory you should have 4 folders:

```
docker@ros2:~/wdir$ ls
build  install  log  src
```

- The `src` directory is where you will put and edit your source code
- 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. Without any option, all the needed files are copied in this directory (binary from `build`, scripts from `src`...). With the `--symlink-install` option, the files are not copied but linked instead (this should be chosen when possible for development).
- The `log` directory contains various logging information about each `colcon` invocation.

Creating a package

> A package can be considered a container for your ROS 2 code. If you want to be able to install your code or share it with others, then you'll need it organized in a package. With packages, you can release your ROS 2 work and allow others to build and use it easily.
> https://docs.ros.org/en/humble/Tutorials/Beginner-Client-Libraries/Creating-Your-First-ROS2-Package.html

One could use the `ros2 pkg create` to create a package. For instance, to create python package with a node inside:

```
ros2 pkg create --build-type ament_python --node-name my_first_node
my_first_package
```

But it seems that in the considered ROS2 version (humble) the `--symlink-install` option does not as expected for python packages. As we are going to use python package, and we do not want to build/install after updating a single line of code, we will do the package "by hand" to be sure that it is configured properly.

A package is just a directory with two files in it : a `package.xml` file and a `CMakeLists.txt` file to use CMake.

Create the package directory

Into the `src` directory of your workspace, create a `my_first_package` directory.

```
docker@ros2:~/wdir/src$ mkdir my_first_package
```

Inside that directory, create a `src` directory and two files `package.xml` and `CMakeLists.txt`:

```
docker@ros2:~/wdir/src$ cd my_first_package/
docker@ros2:~/wdir/src/my_first_package$ mkdir src
docker@ros2:~/wdir/src/my_first_package$ > package.xml
docker@ros2:~/wdir/src/my_first_package$ > CMakeLists.txt
```

You then should have:

```
.
└── wdir
    ├── build
    ├── install
    ├── log
    └── src
        └── my_first_package
            ├── CMakeLists.txt
            ├── package.xml
            └── src
```

CMakeLists.txt

Here is the content of the `CMakeLists.txt` file. This file is needed for `CMake` to build the package.

```
cmake_minimum_required(VERSION 3.5)
project(my_first_package) # The name of the package

# ament_cmake is the build system for CMake based packages in ROS 2,
https://docs.ros.org/en/humble/How-To-Guides/Ament-CMake-Documentation.html
find_package(ament_cmake REQUIRED)
find_package(ament_cmake_python REQUIRED) # as we are doing python nodes

file(GLOB scripts src/*) # to process all the py scripts in the src
directory at once (instead of having an install line for each script)
install(PROGRAMS ${scripts} DESTINATION lib/${PROJECT_NAME}) # to install
the scripts

ament_package()
```

package.xml

Here is the content of the `package.xml` file. This file is needed to set up the ROS2 package.

```
<?xml version="1.0"?>
<?xml-model href="http://download.ros.org/schema/package_format3.xsd"
schematypens="http://www.w3.org/2001/XMLSchema"?>
<package format="3">
  <name>my_first_package</name> <!-- NAME OF THE PACKAGE -->
  <version>1.0.0</version>
  <description>Description of the package</description> <!-- DESCRIPTION OF
THE PACKAGE-->
  <maintainer email="me@my.mail">My Name</maintainer> <!-- YOUR NAME AND
MAIL -->
  <license>MIT</license> <!-- LICENSE FOR THE PACKAGE -->
```

```
  <!-- ament_cmake is the build system for CMake based packages in ROS 2,
https://docs.ros.org/en/humble/How-To-Guides/Ament-CMake-Documentation.html
-->
  <buildtool_depend>ament_cmake</buildtool_depend>

  <export>
    <build_type>ament_cmake</build_type>
  </export>
</package>
```

You can try to build the package. In the `wdir` directory do the command:

```
docker@ros2:~/wdir$ colcon build
Starting >>> my_first_package
Finished <<< my_first_package [3.45s]

Summary: 1 package finished [4.11s]
```

You can also check that in the `install` directory you have the `my_first_package` directory now.

Basic node

We have a package but no node available... Let's create one.

Create a new file named `my_first_node.py` in the `src` directory of your package:

```
docker@ros2:~/wdir/src/my_first_package/src$ > my_first_node.py
```

Write a basic python hello world:

```
#!/bin/python3

def main():
    print('Hi from my_first_package.')

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

> For a python script to be seen as an executable and executed properly you must
>
> - add a shebang to specify that the script must be executed with python3 (the first line should be `#!/bin/python3`, https://en.wikipedia.org/wiki/Shebang_(Unix))
> - set the script to be executable:
>
> ```
> chmod +x my_first_node.py
> ```

You can build the package again (be careful to be in the `wdir` directory!)

```
docker@ros2:~/wdir$ colcon build
Starting >>> my_first_package
Finished <<< my_first_package [2.99s]

Summary: 1 package finished [4.04s]
```

and now in the `install` directory you should have the executable:

```
.
├── build
│   └── ...
├── install
│   ├── ...
│   ├── my_first_package
│   │   ├── lib
│   │   │   └── my_first_package
│   │   │       └── my_first_node.py
│   │   └── share
│   │       ├── ...
...
```

```
docker@ros2:~/wdir$ ls -al install/my_first_package/lib/my_first_package/
total 12
drwxr-xr-x 2 docker docker 4096 Nov 21 14:31 .
drwxr-xr-x 3 docker docker 4096 Nov 21 14:31 ..
-rwxr-xr-x 1 docker docker   89 Nov 21 14:25 my_first_node.py
```

Running the node

Try to execute the node:

```
docker@ros2:~/wdir$ ros2 run my_first_package my_first_node.py
Package 'my_first_package' not found
```

It fails because your custom package is not known from ros2 command. You have to say to ROS2 that you have packages and nodes in your workspace. To do that you have to add your workspace to the path:

```
docker@ros2:~/wdir$ pwd
/home/docker/wdir
docker@ros2:~/wdir$ source install/setup.bash
```

Now this should run correctly :

```
docker@ros2:~/wdir/src$ ros2 run my_first_package my_first_node
Hi from my_first_package.
```

Updating the node

Try to update the message displayed by the node. For instance:

```
def main():
    print('Hi from my_first_package. Test number 2.')
```

And run the node again, you should have :

```
docker@ros2:~/wdir/src$ ros2 run my_first_package my_first_node
Hi from my_first_package.
```

The executable has not been updated in the `install` directory: as mentioned earlier, by default the files/scripts/binaries are copied into the `install` directory. To avoid that, you can build your package with the `--symlink-install` option:

```
docker@ros2:~/wdir$ colcon build --symlink-install
Starting >>> my_first_package
Finished <<< my_first_package [2.09s]

Summary: 1 package finished [2.99s]
```

If you check in your `install` directory, now you can see that the files are not copied, but linked instead:

```
docker@ros2:~/wdir$ ls -al install/my_first_package/lib/my_first_package/
total 8
drwxr-xr-x 2 docker docker 4096 Nov 21 14:43 .
drwxr-xr-x 3 docker docker 4096 Nov 21 14:31 ..
lrwxrwxrwx 1 docker docker   59 Nov 21 14:43 my_first_node.py ->
/home/docker/wdir/src/my_first_package/src/my_first_node.py
```

Now you can try to update the node without building the package: it should update the execution as well!

Creating a publisher

Create a new file named `my_publisher.py` in the `src/my_first_package/my_first_package/` directory (next to the `my_first_node.py` file) with the following content :

```
#!/bin/python3

def main():
    print('Hi from my publisher node.')

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

Now we would like to add this file to our package. To do that, just make the file executable and build the package again:

```
docker@ros2:~/wdir/src/my_first_package/src$ chmod +x my_publisher.py
```

> Note: you may need to remove by hand the `install`, `build` and `log` directory to do a clean build

You should then have:

```
docker@ros2:~/wdir$ ls -al install/my_first_package/lib/my_first_package/
total 8
drwxr-xr-x 2 docker docker 4096 Nov 21 14:53 .
drwxr-xr-x 3 docker docker 4096 Nov 21 14:53 ..
lrwxrwxrwx 1 docker docker   59 Nov 21 14:53 my_first_node.py ->
/home/docker/wdir/src/my_first_package/src/my_first_node.py
lrwxrwxrwx 1 docker docker   58 Nov 21 14:53 my_publisher.py ->
/home/docker/wdir/src/my_first_package/src/my_publisher.py
```

And you should be able to run the executable:

```
docker@ros2:~/wdir$ ros2 run my_first_package my_publisher.py
Hi from my publisher node.
```

Update the file to implement a simple publisher:

```
#!/bin/python3

# import ROS libraries
import rclpy
from rclpy.node import Node

# imports the built-in string message type
# it will the type of the published messages
```

```
from std_msgs.msg import String

class MinimalPublisher(Node):
    # The MinimalPublisher class inherits from the Node class

    publisher_: rclpy.publisher.Publisher  # publisher to publish the
message
    timer: rclpy.timer.Timer               # timer to send the message
periodically
    i: int                                 # counter for the index of
messages

    def __init__(self):
        # constructor of the class

        # call the upper class constructor (the Node constructor)
        # the str as argument is to define the node name when running (ros2
node list)
        super().__init__('minimal_publisher')

        # create a publisher to publish the messages
        #   String : the type of message (imported from std_msgs.msg)
        #   'myTopic' : the topic name to send the messages
        #   10 : the “queue size” is 10.
        #       Queue size is a required QoS (quality of service) setting
that limits the amount of queued messages
        #       if a subscriber is not receiving them fast enough.
        self.publisher_ = self.create_publisher(String, 'myTopic', 10)

        # A timer is created with a callback to execute every 0.5 seconds.
        # self.i is a counter used in the callback.
        timer_period = 0.5  # 0.5 second -> 500ms
        self.timer = self.create_timer(timer_period, self.timer_callback)
        self.i = 0

    def timer_callback(self):
        # timer_callback creates a message with the counter value appended,
        # and publishes it to the console with get_logger().info.
        msg = String()
        msg.data = f'Hello World: {self.i}'
        self.publisher_.publish(msg)  # publish the message
        # log with [INFO] flag
        self.get_logger().info(f'Publishing: "{msg.data}"')
        self.i += 1


def main(args=None):
    # to init the ros interface
    rclpy.init(args=args)

    # creating the node from our custom class
    minimal_publisher = MinimalPublisher()

    # start the node and loop until it closes
```

```
    rclpy.spin(minimal_publisher)

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


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

Build the package and run the node. You should have something like:

```
docker@ros2:~/wdir$ ros2 run my_first_package my_publisher.py
[INFO] [1678894570.129581422] [minimal_publisher]: Publishing: "Hello
World: 0"
[INFO] [1678894570.565274689] [minimal_publisher]: Publishing: "Hello
World: 1"
[INFO] [1678894571.065611245] [minimal_publisher]: Publishing: "Hello
World: 2"
[INFO] [1678894571.565082164] [minimal_publisher]: Publishing: "Hello
World: 3"
[...]
```

From a new prompt you can check if the message is indeed published:

```
docker@ros2:~/wdir$ ros2 topic echo /myTopic
data: 'Hello World: 30'
---
data: 'Hello World: 31'
```

Creating a subscriber

Create a new file named `my_subscriber.py` with the following content:

```
#!/bin/python3

import rclpy
from rclpy.node import Node

from std_msgs.msg import String


class MinimalSubscriber(Node):
```

```
    subscription: rclpy.subscription.Subscription

    def __init__(self):
        super().__init__('minimal_subscriber')
        self.subscription = self.create_subscription(
            String, # the message type
            'myTopic', # the topic name
            self.listener_callback, # function called when getting a
message
            10)

    def listener_callback(self, msg:String):
        self.get_logger().info(f'I heard: "{msg.data}"')


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

    minimal_subscriber = MinimalSubscriber()

    rclpy.spin(minimal_subscriber)

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


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

Add it to the package, build, and run the node. When running the publisher and the subscriber at the same time you should have the following output for the subscriber :

```
docker@ros2:~/wdir$ ros2 run my_first_package my_subscriber
[INFO] [1678895446.591535782] [minimal_subscriber]: I heard: "Hello World:
1557"
[INFO] [1678895447.077905812] [minimal_subscriber]: I heard: "Hello World:
1558"
[INFO] [1678895447.577434319] [minimal_subscriber]: I heard: "Hello World:
1559"
[...]
```

Avoiding error

Note that you can have error when stopping the node with the Ctrl+C terminal command. To avoid that error you can:

- Catch the `KeyboardInterrupt` error;

- Comment the `rclpy.shutdown()` as it seems to be already done when doing Ctrl+C.

```
    # start the node and loop until it closes
    try:
        rclpy.spin(minimal_publisher)
    except KeyboardInterrupt as ki:
        pass

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