Compiling & Writing Your First ROS2 Nodes: Talker and Listener

Table of Contents

Introduction

Objective:

Learn how to use Docker to compile ROS2 code and create a custom ROS2 node that transforms some simple data.

In this lab we will practice using Docker to compile our code and write our first ROS2 node. Here's a general outline of the steps we will work through:

1. Pull source code for the lab from Github
2. Use a *Makefile* to mount our local ROS2 workspace into the Docker container's filesystem
3. Use the ROS2 CLI to compile and run some example nodes: *talker* and *listener*
4. Write & compile your own nodes that will square the data sent from the talker

Note:
The code you clone will be saved locally on your computer, but *you likely do not have ROS2 installed natively* on your computer. Instead, we'll use:

- Docker → provides a pre-configured environment with ROS2 installed
- Makefiles → simplifies building/running the container while mounting your local code

This, way we can develop ROS2 nodes without needing a full ROS2 setup on our host systems! Finally, note that the Makefile will handle most of the necessary Docker commands for you. I encourage you to examine the Makefile and explore how it works exactly.

Keywords:

Some key words used in this document and their meaning:
Host system – your laptop, "host computer"

Getting Started

Cloning source code from CPE416 GitHub

First, we'll need to clone the repository that I have prepared for the class. Run the following command in a terminal on your *host system* (take note of the directory you're in when executing this)

git clone https://github.com/ambulantelab/CPE416.git

Now that we've cloned the repository locally on your host, we'll use the provided *makefile* to mount our local copy into the Docker container. This allows the container to access your local folder – meaning you can edit the code on any host program but compile and execute the same code inside the container.

Before we can start using the makefile, we'll need to create a *docker network.* To do this we can run the following command (this creates a docker network named 'ros' which the makefile expects):

```
docker network create ros
```

Editing the Makefile:

In this portion of the lab, we'd want to edit the provided Makefile so that when you run the *make bash* command the correct *image* is used, and a container instance is created.

1. Update the Container Image Reference

Locate this line in the Makefile you have, after cloning the repository

```
CONTAINER_NAME = ??
```

Replace it with the name of the Ubuntu ROS2 *image* that is compatible with your host system. For example, for my host system (MacBook M2) I have pointed it to the *arm64v8/ros:humble* image that I would like to use:

```
CONTAINER_NAME = arm64v8/ros:humble
```

2. Use the Makefile to Create a New Container

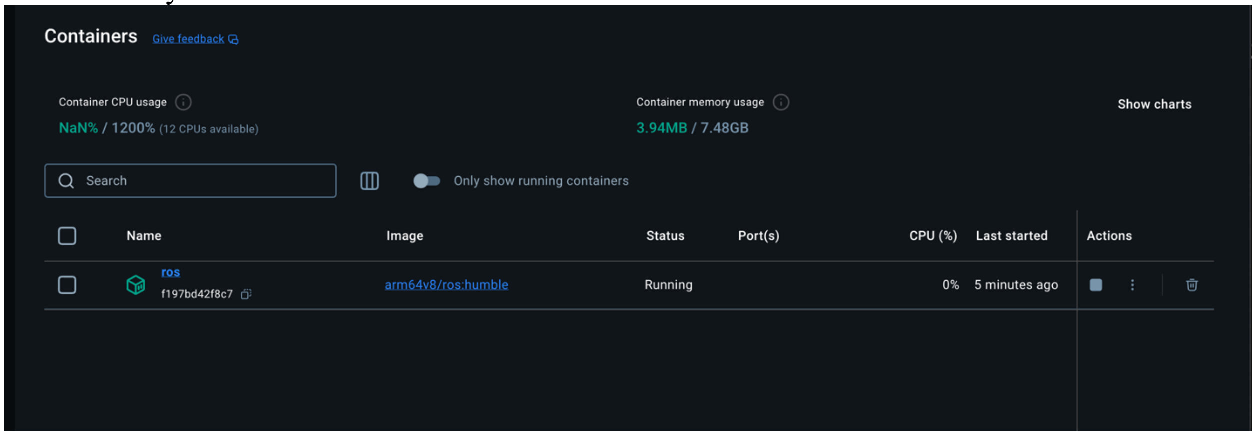
From inside a terminal in the Docker Desktop application you can run the command. (Make sure you're in the directory where the *makefile* is located.):

```
make bash NAME=YOUR_NAME_HERE
```

This last command will:

- Create a new container instance from the specified image
- Mount your local repo into the container (*/CPE416*)
- Launch an interactive bash shell inside the container

You should replace "YOUR_NAME_HERE" with any container name you prefer (like "ros" for simplicity). You'll know it worked correctly when you see your named container appear in the Docker Desktop's *Containers* list – don't worry if other containers appear alongside it, as this won't affect your work. The screenshot below shows an example where we named the container "ros" for easy identification.



Container CPU usage
NaN% / 1200% (12 CPUs available)

Container memory usage
3.94MB / 7.48GB

Name	Image	Status	Port(s)	CPU (%)	Last started	Actions
ros f197bd42f8c7	arm64v8/ros:humble	Running		0%	5 minutes ago	

Figure 1: A screenshot of the docker dashboard after running the make bash command.

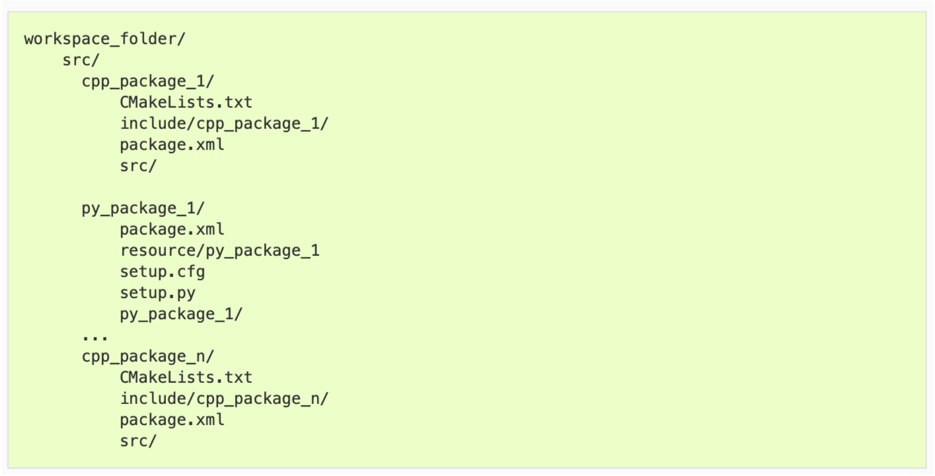
Creating and Compiling an Empty ROS2 Workspace

Now that your terminal is active inside the Docker container, you'll need to navigate to the project directory. Run the `cd /CPE416/` command to enter the workspace folder – if you're unsure of your current location, use the `ls` command to list directory contents and verify you're in the correct starting position before trying to change directories. Remember, you're now working in the container's Linux environment, so all paths are case-sensitive and follow Linux conventions.

Create and compile an empty ROS2 workspace

1. Now, let's set up the proper directory structure. From your terminal inside the container (while in the */CPE416/* directory) run:

```
mkdir -p ros_ws/src
```



```
workspace_folder/
    src/
      cpp_package_1/
          CMakeLists.txt
          include/cpp_package_1/
          package.xml
          src/

      py_package_1/
          package.xml
          resource/py_package_1
          setup.cfg
          setup.py
          py_package_1/
      ...
      cpp_package_n/
          CMakeLists.txt
          include/cpp_package_n/
          package.xml
          src/
```

Figure 2: What the folder structure looks like for a typical ros workspace. Keep this in mind when creating packages and running nodes. Note many of the above files are auto-generated by ROS commands.

This single command creates both the ROS2 workspace (*ros_ws)* and its required source directory *(src)* in one step, with the correct nested structure (*/CPE416/ros_ws/src/*).

2. Next, compile the empty workspace by navigating to the *ros_ws* directory and running the appropriate command. **Note:** the command is omitted here intentionally! Investigate for yourself the ROS2 documentation and resources for the correct command.

After running the command, you should see CMake output confirming your workspace is properly initialized, even without and packages yet. The directory structure should now be to look like the reference image, ready for adding ROS2 packages later. **Note:** the build command **should** be run from the workspace root directory i.e. *ros_ws*

Understanding the Workspace Structure

After running the compilation command from your *ros_ws* directory, you'll see three generated folders:

- *build/* - Contains intermediate build artifacts
- *install/* - Holds the compiled packages and setup files
- *log/* - Stores detailed build logs

Since your *src/* folder is empty, the build will complete quickly without processing any packages.

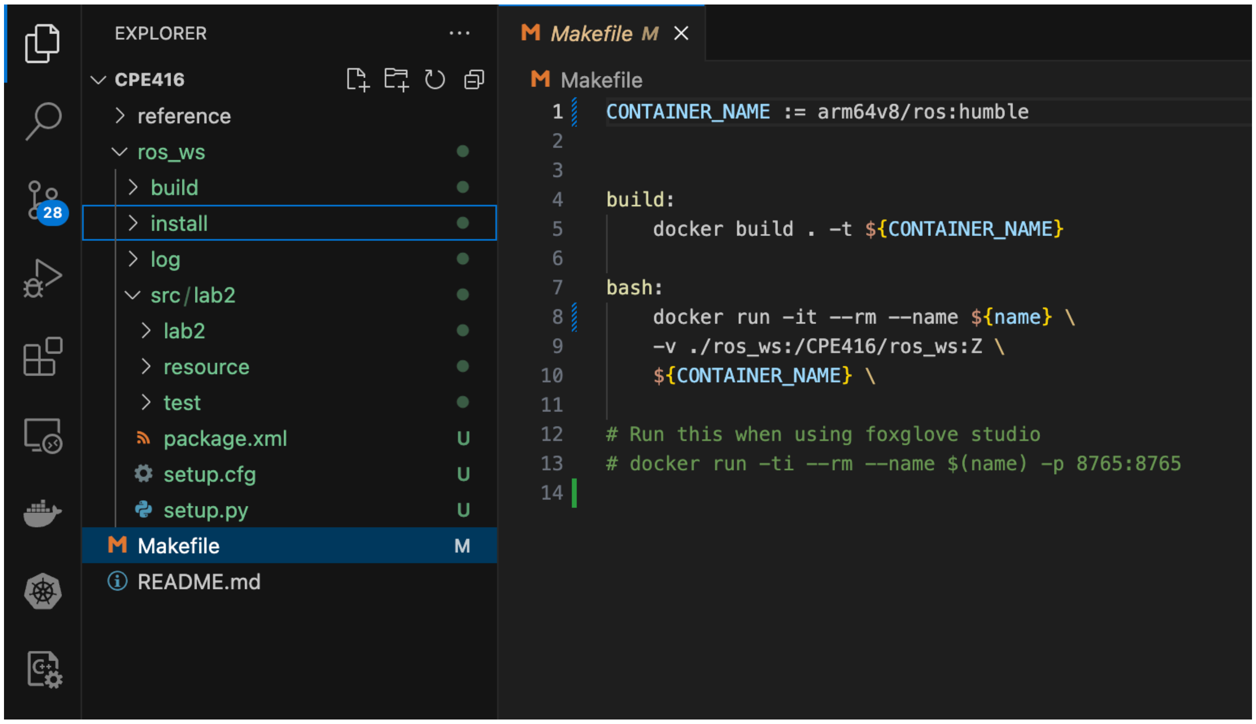
Create an empty ROS2 Python package in your workspace.

From inside your workspace's *src/* directory, create a Python package by using the ROS2 CLI as follows:

```
ros2 pkg create --build-type ament_python lab1
```

For this lab we are going to name the package *lab1*. (Thus, in the command above name the package *lab1*). This command will create the package and provide you with many necessary files that *are required* for building your code.

Look at the screenshot below for an idea of what the contents of your new *lab1* package should be. **Note:** package name should follow ROS2 conventions – lowercase, underscores



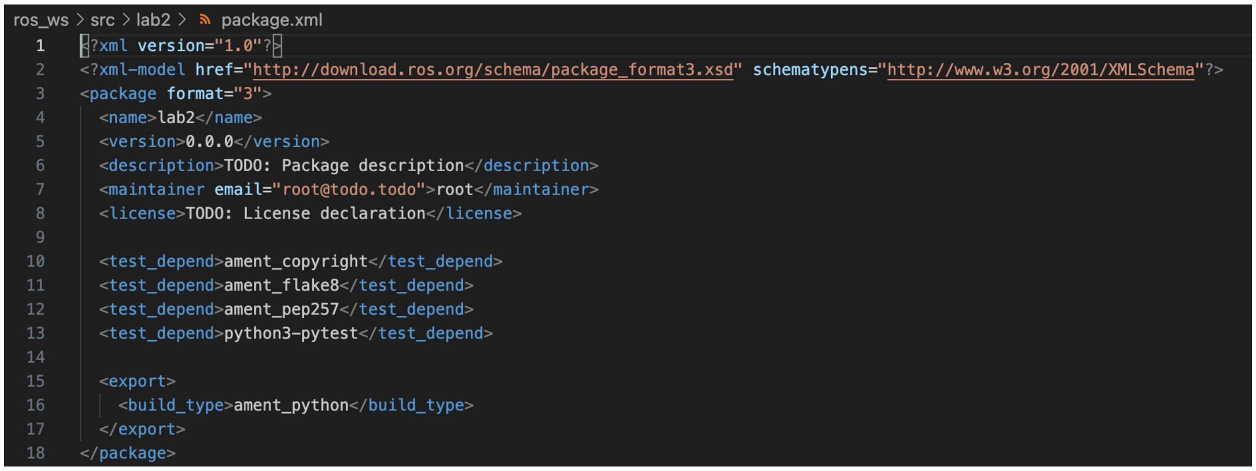
```
CONTAINER_NAME := arm64v8/ros:humble


build:
	docker build . -t ${CONTAINER_NAME}

bash:
	docker run -it --rm --name ${name} \
	-v ./ros_ws:/CPE416/ros_ws:Z \
	${CONTAINER_NAME} \

# Run this when using foxglove studio
# docker run -ti --rm --name $(name) -p 8765:8765

```

Figure 3: What your folders should look like after you've run the ros2 pkg create command. Notice how many files and folders where auto generated. Application running is VisualCode. (You can ignore the 'lab2' name for the current lab).

Key Configuration Files for your ROS2 Package

Two auto-generated files will require modification to ensure proper compilation:

package.xml



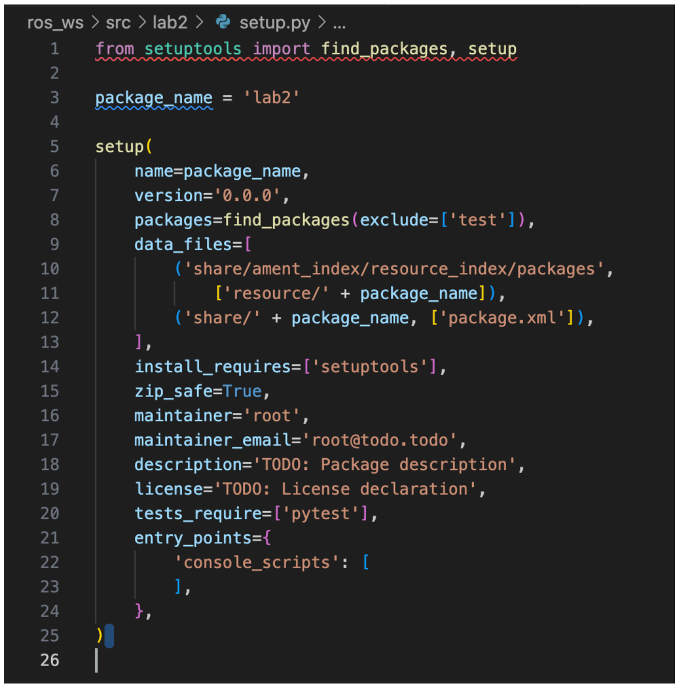
ros_ws > src > lab2 > package.xml

```xml
<?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>lab2</name>
  <version>0.0.0</version>
  <description>TODO: Package description</description>
  <maintainer email="root@todo.todo">root</maintainer>
  <license>TODO: License declaration</license>

  <test_depend>ament_copyright</test_depend>
  <test_depend>ament_flake8</test_depend>
  <test_depend>ament_pep257</test_depend>
  <test_depend>python3-pytest</test_depend>

  <export>
    <build_type>ament_python</build_type>
  </export>
</package>
```

Figure 4: package.xml files that was auto generated by ros command.

- Purpose: Declares packages dependencies (i.e. other ROS2 packages/libraries your code requires)
- Location: *~/CPE416/ros_ws/src/lab1/package.xml*

setup.py



ros_ws > src > lab2 > setup.py > ...

```python
from setuptools import find_packages, setup

package_name = 'lab2'

setup(
    name=package_name,
    version='0.0.0',
    packages=find_packages(exclude=['test']),
    data_files=[
        ('share/ament_index/resource_index/packages',
            ['resource/' + package_name]),
        ('share/' + package_name, ['package.xml']),
    ],
    install_requires=['setuptools'],
    zip_safe=True,
    maintainer='root',
    maintainer_email='root@todo.todo',
    description='TODO: Package description',
    license='TODO: License declaration',
    tests_require=['pytest'],
    entry_points={
        'console_scripts': [
        ],
    },
)
```

Figure 5: setup.py file that was auto-generated by ros command.

- Purpose: Defines package entry points (node executables) and installation structure
- Location: *~/CPE416/ros_ws/src/lab1/setup.py*

We need to edit these files to ensure proper code compilation. Note that the *ros2 pkg create* command automatically created a *lab1* folder inside your package directory – this is where your node's source code should be placed. The *package.xml* file specifies all package dependencies, while *setup.py* directs the compiler to your source code and defines the names of the compiled executable files.

Finally, compile the empty package *lab1* inside of your workspace. You should see CMake output indicating the package compiled successfully.

Creating and Compiling a Custom ROS2 Node

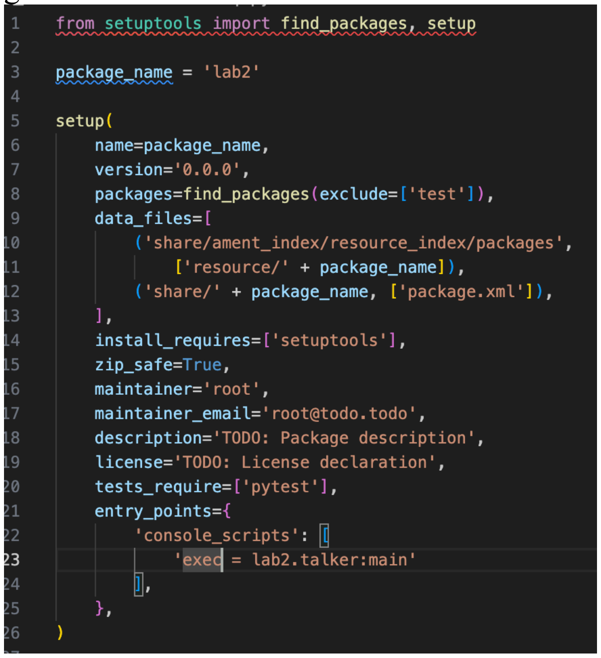
First, let's migrate the existing code from the repository into our new package. Inside the *ros_ws/src/lab1/lab1/* directory, create two files: *talker.py* (for the publisher) and *listener.py* (for the subscriber). Copy the contents of *pub-sub/publisher.py* into *talker.py*, and similarly paste *pub-sub/subscriber.py* into *listener.py*. This places our node implementations in the correct package structure.

Listing Nodes for Compilation

To enable compilation, we need to modify the *entry_points* dictionary in *setup.py (see screenshots in previous section)*. Locate the *console_scripts* list within this dictionary - this is where we define our executable targets. Add entries using the syntax:

```
executable_name = package_name.source_file:main
```

For example, if I wanted to compile a node named *talker* from a package named *lab2* and generate an executable file *exec* then I would write the following:



```python
from setuptools import find_packages, setup

package_name = 'lab2'

setup(
    name=package_name,
    version='0.0.0',
    packages=find_packages(exclude=['test']),
    data_files=[
        ('share/ament_index/resource_index/packages',
            ['resource/' + package_name]),
        ('share/' + package_name, ['package.xml']),
    ],
    install_requires=['setuptools'],
    zip_safe=True,
    maintainer='root',
    maintainer_email='root@todo.todo',
    description='TODO: Package description',
    license='TODO: License declaration',
    tests_require=['pytest'],
    entry_points={
        'console_scripts': [
            'exec = lab2.talker:main'
        ],
    },
)
```

When configuring your setup.py, it's crucial to understand these distinct elements:

- Executable Name (*exec* in the example above):
    - This becomes the command you'll run to start the ROS2 node (e.g., *ros2 run lab1 exec*)
    - Defined in the left side of the *console_scripts* entry
- Package Path (*lab1.talker:main*):
    - Specifies which function to execute:
    - *lab2* → Your Python package
    - *talker* → The .py file (without extension)

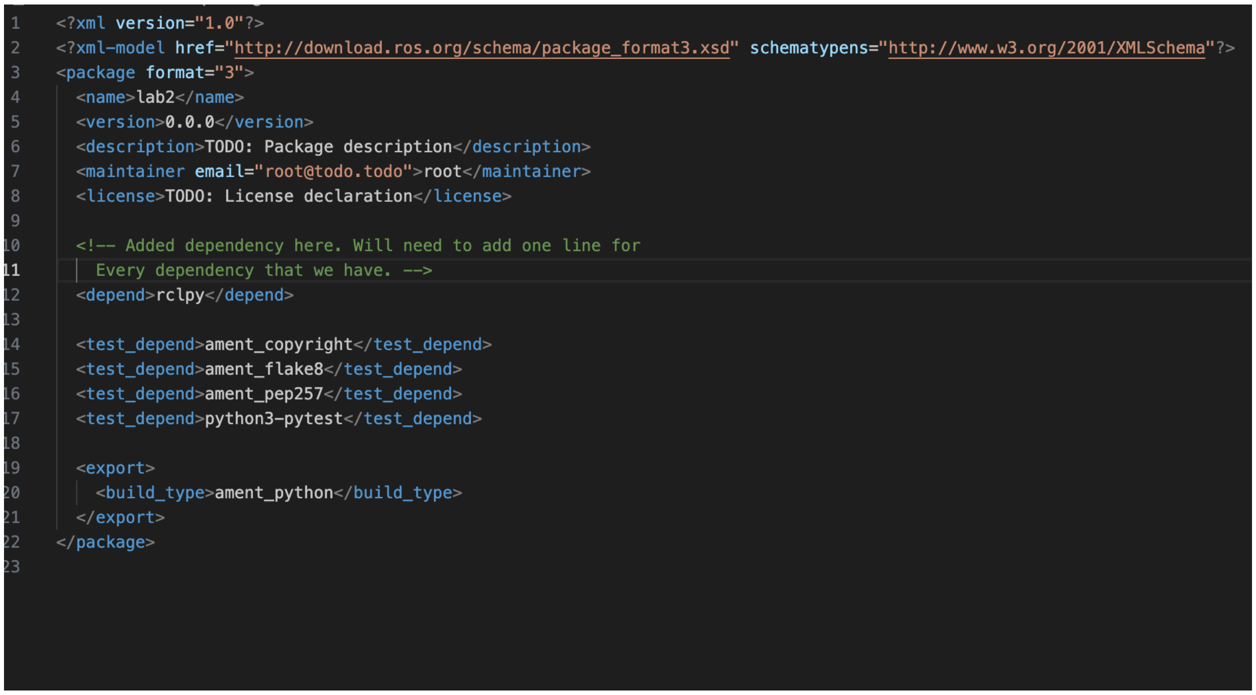
    - *main* → The function to call (can be renamed if needed)
- Node Name (e.g., *minimal_publisher*):
    - Defined inside your Python code via *rclpy.create_node()*
    - What appears when you run the command *ros2 node list*

Important Distinction: These three elements (executable name, source file, and node name) are independent. While you can name them identically (e.g., all "talker"), keeping them, distinct helps manage complex projects.

For example (imagine two different Python files):
```
# setup.py
entry_points = {'console_scripts': ['exec = lab2.talker:main']}

# talker.py (different file from above)
def main():
  node = rclpy.create_node('minimal_publisher') # <- node name
```

Adding Dependencies for our Package

Since our node uses *rclpy* (ROS2's Python client library) and the *std_msgs* module, we must declare these dependencies in the *package.xml* file. Open the file and locate the <*depend*> tags section. Add:
```
<depend>rclpy</depend>
<depend>std_msgs</depend>
```


```
<?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>lab2</name>
  <version>0.0.0</version>
  <description>TODO: Package description</description>
  <maintainer email="root@todo.todo">root</maintainer>
  <license>TODO: License declaration</license>

  <!-- Added dependency here. Will need to add one line for
    Every dependency that we have. -->
  <depend>rclpy</depend>

  <test_depend>ament_copyright</test_depend>
  <test_depend>ament_flake8</test_depend>
  <test_depend>ament_pep257</test_depend>
  <test_depend>python3-pytest</test_depend>

  <export>
    <build_type>ament_python</build_type>
  </export>
</package>
```

Important Note: missing dependencies cause the *ImportError* at runtime!

You only need to do this once per dependency for the whole package. If you create another node, you'll need to update *setup.py*, but **not** *package.xml* **if** you don't have any new dependency.

Package and Node Compilation

Now that your package is configured, compile it by running the build command from your *ros_ws/* directory. After successful compilation, you must source your workspace for ROS2 to recognize your new nodes. This is achieved with the following command:

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

Important Note: The above command (as is) would need to be run from the root directory of your workspace (*ros_ws). This step is crucial as ROS won't find your executables without it.

To verify your package is properly registered, check ROS2's package list command:

```
ros2 pkg list | grep lab2
```

you should see *lab1* output in your terminal, confirming successful installation.

Running a Node ROS2 CLI

Execute your node

Finally, after compilation of your package has completed successfully, you can run your node with the following ROS2 CLI command:

```
ros2 run lab1 exec
```

You should see the following live output after running the command:

```
[INFO] [1736278731.198436512] [minimal_publisher]: Publishing: "Hello World: 0"
[INFO] [1736278731.684297387] [minimal_publisher]: Publishing: "Hello World: 1"
```

This demonstrates a working publisher node sending incrementing messages to a topic.

Verify Topic Creation

In a new terminal session, connect to your running Docker container (named "ros" in this example):

```
docker exec -it ros bash
```

Source the ROS2 environment and check active topics:

```
source /opt/ros/humble/setup.bash
ros2 topic list
```

You should see */my_topic* listed, confirming successful topic creation.

Code Structure

Please study the code in *talker.py* and learn how the functions and classes work. ROS2 coding should be object orientated. Notice how after inheriting from the *rclpy node* class we have access to functions that create: publishers, timers, subscribers and callbacks.

Key OOP patterns to observe:

- **Inheritance:** The node class extends the *rclpy.Node* class
- **Encapsulation:** Publisher/timer logic is contained within the class
- **Callback Methods (more later):** Notice how callbacks handle events
- **Modularity:** Separation of node initialization and business logic

Task: Repeat previous steps to run a listener node:

Edit the *setup.py* file again to add another executable (the listener). Run the node at the same time as the talker to make sure that they are correctly connected. Note you will need two terminals for this. One running the *talker* and the other running the *listener*.

Assignment: Listener Squared

1. Write a node, talker, that publishes integers to a topic named */numbers*. (The type of the topic should ***NOT*** be *std_msgs/String*).
2. Write another node that subscribes to the topic */numbers* and prints out to the console the **squared number** of the message that came over topic.
3. Let's keep the executable names consistent, name the first node executable *talker* and the second node executable *squared*

Testing & Debugging

Note that testing your ROS2 code, in general, can be difficult. Remember to use the ROS2 command line tools to check the topics that are present when nodes are running. Some Useful command for this:

```
ros2 topic list # Check all active topics
ros2 topic info /numbers # Verify topic type and connections
ros2 topic echo /numbers # Inspect message content
```

Common Pitfall: A single typo in topic names (e.g. */my_topic* vs */mytopic*) will silently impede and connections between publisher and subscriber.

Make sure that your *package.xml* and *setup.py* files are correct, otherwise your code will not compile. Missing dependencies can cause import failures, and incorrect entry points prevent node execution.

When logs aren't enough, you can also try Python's debugger (to step through your code):

```
import pdb; pdb.set_trace()# Add this where you need inspection
```

Debugger commands:
n (next), *s* (step into), *c* (continue), *l* (list code)
print(variable) to check values

Submission

Code Demos. You are expected to demo your nodes and code in the next lab section. Make sure that I get your name down after your successful demo.

After you've demo'ed your code and passed, I will need you to submit your source code.

1. Delete the: *build*, *log*, and *install* folders in your ros workspace. (I do not need to check your compilation outputs.)
2. Compress your ros workspace and upload the compressed file to Canvas. For example, a zip file would work. This compressed file should contain your source code.