**ROS 2 Python package creation**

**ROS2 Tutorials youtube page:**

**https://www.youtube.com/@RoboticsBackEnd**

ROS 2 python package creation is different from traditional ROS1 package creation. Here are the steps needed to follow in order to create a ROS 2 package.

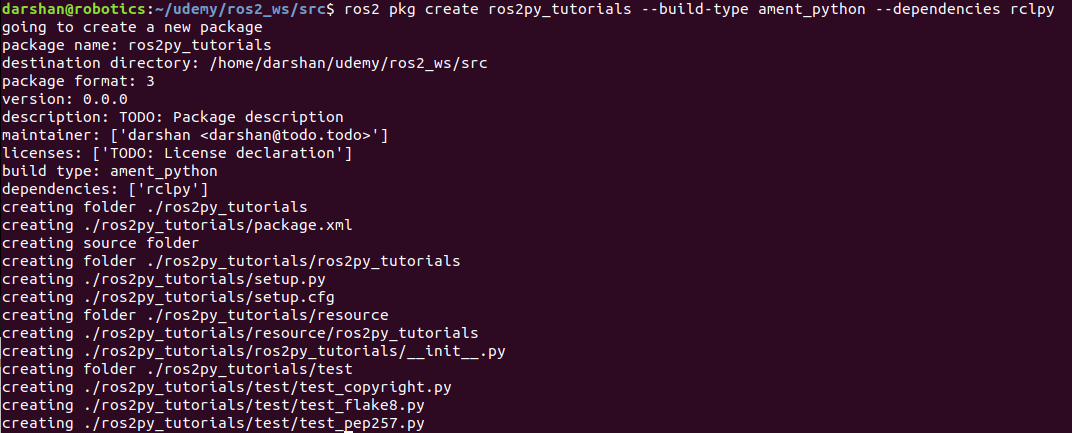
**Step 1: Create python package**

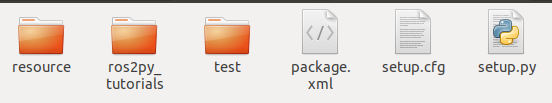
Create dummy ROS 2 python package on terminal

$ ros2 pkg create <package\_name> --build-type ament\_python --dependencies rclpy

$ ros2 pkg create ros2py\_tutorials --build-type ament\_python --dependencies rclpy

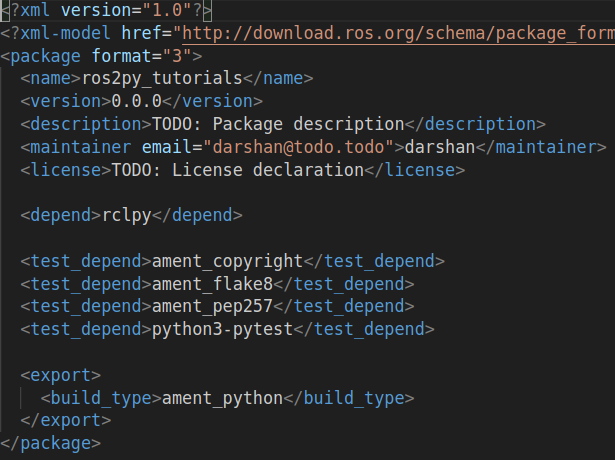
This will create a package which comes with setup.py & package.xml files. Here **setup.py** is the replacement file of CMakelist for python projects.





Keep all nodes inside the ros2py\_tutorials folder.

Package.xml file



**Note:** Since compiling python scripts, build\_type should be **“ament\_python”.**

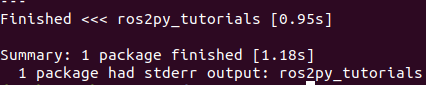
<build\_type>ament\_python</build\_type>

<depend>rclpy</depend>

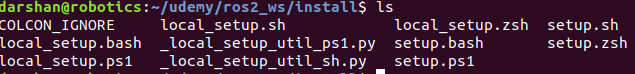
**Step2 : Build the package:**

$ cd ~ros2\_ws

$ colcon build



Here inside install folder you can see



local\_setup.bash → This bash file helps to source only this workspace to the terminal.

setup.bash →This bash file helps to source this workspace + global opt ros2 files to the terminal.

**Building particular package:**

colcon build --packages-select ros2py\_tutorials --symlink-install

**Note:** Now this python package is ready to host any ros2 python nodes.

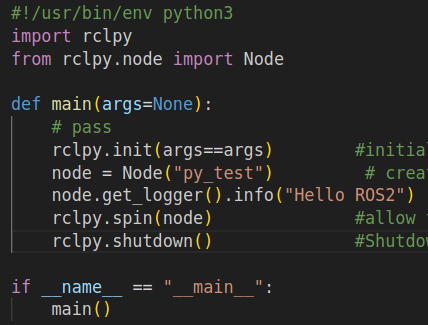
**Step 3: Create a minimal python node script**

Get inside the package and name of folder which having the same name as package[where python init.py file located]

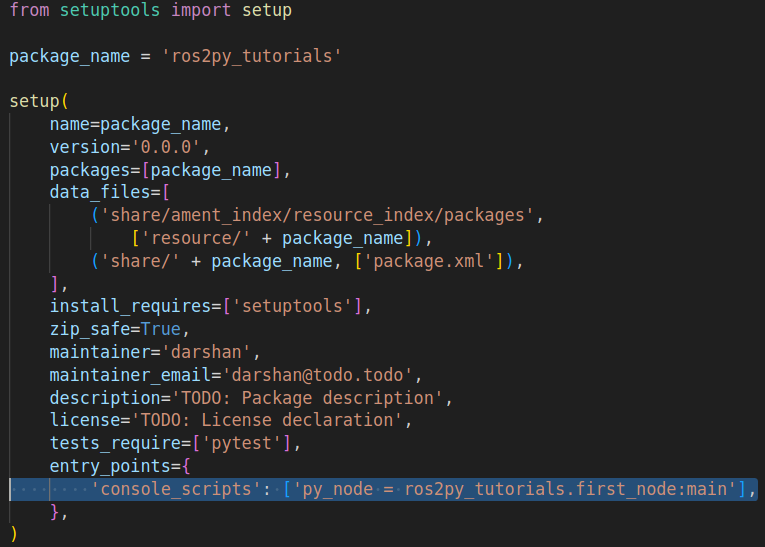
$ cd ~ros2\_ws

$ cd src/ros2py\_tutorials/ros2py\_tutorials

$ code first\_node.py



**Step 4** : Edit setup.py file for ros2 node executables



Note: Edit entry\_points variable,

entry\_points = {‘console\_scripts’ : ['**testpy\_node = ros2py\_tutorials.first\_node:main'**] }

Where ,

testpy\_node → is the name of executable (optional, you can give preferable one)

ros2py\_tutorials → name of package

.first\_node → name of file

:main → name of function need to execute inside script

**Compile the package inorder to reflect node executable**

$ colcon build

( or )

$ colcon build --packages-select ros2py\_tutorials

**Note:** After successful compilation you have to find the python executable inside the workspace **install** folder

$ ros2\_ws$ cd install/ros2py\_tutorials/lib/ros2py\_tutorials

$ ls



If this executable is not found inside the install folder means “**the package has not successfully compiled python node**”.

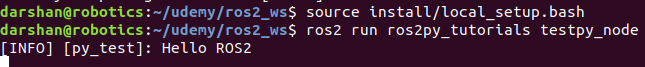
In that cases compile the specific package alone which having python node file by using following command

$ colcon build --packages-select <package\_name>

**Step 5:** Source workspace on terminal and Run executable ros2 node

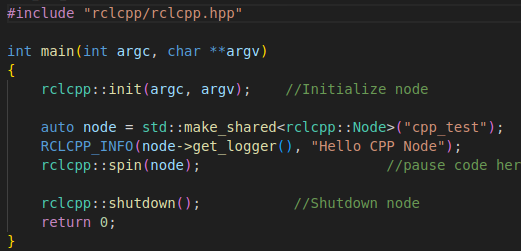
$ source install/local\_setup.bash

$ ros2 run ros2py\_tutorials testpy\_node

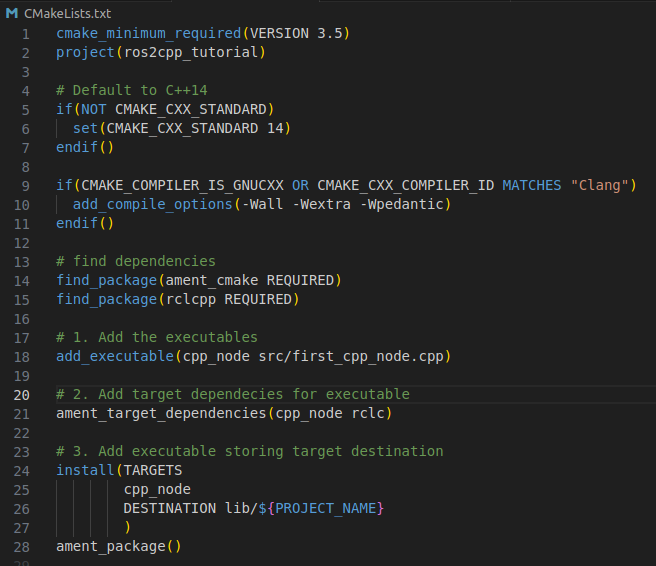


**ROS2 CPP node creation:**

Open src folder inside package and create a ros2cpp file



Edit CMakeList file



Note: 3 things need to edit inside CMakeList file

1. Add executable for ros2cpp node
2. add\_executable(cpp\_node src/first\_cpp\_node.cpp)

Where,

cpp\_node → name of executable (its optional user )

src/first\_cpp\_node.cpp → location of ros2cpp script

2. Add dependencies libraries required to compile node

ament\_target\_dependencies(cpp\_node rclcpp)

cpp\_node → name of executable

rclcpp → libraries required to compile

1. Provide the destination to locate executable object file

install(TARGETS

cpp\_node

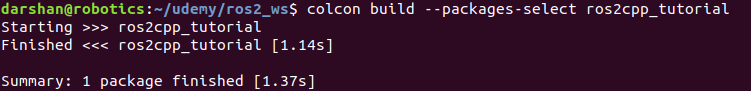
DESTINATION lib/${PROJECT\_NAME}

)

The executable file will be stored inside install/package\_name/lib/package\_name

**Compile ros2cpp node**

$ colcon build --packages-select ros2cpp\_tutorial

****

After successful compilation search for executable file



**Run ros2cpp node**

$ source install/local\_setup.bash

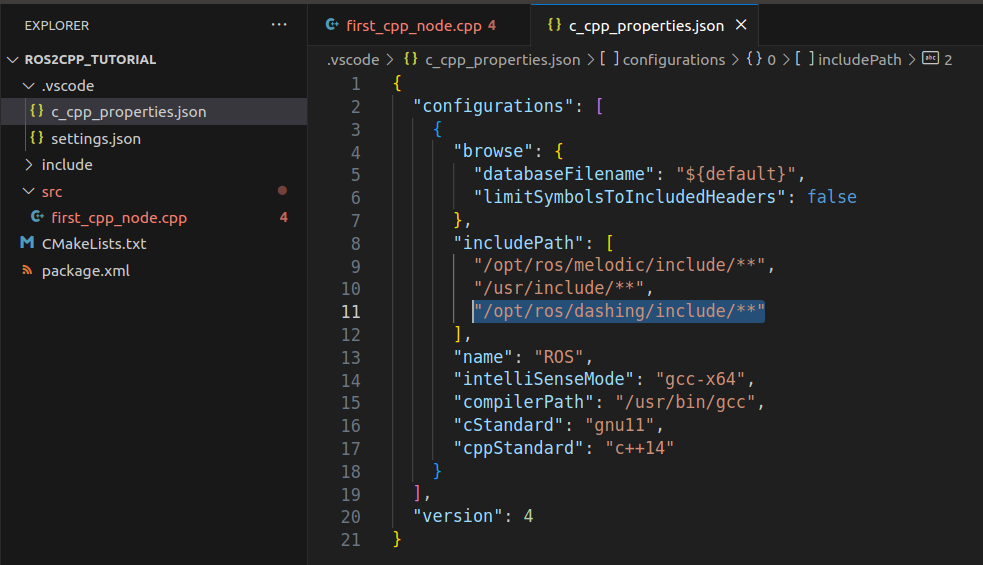
$ ros2 run ros2cpp\_tutorial cpp\_node



**ERROR:** Include rclcpp header file with vscode configuration

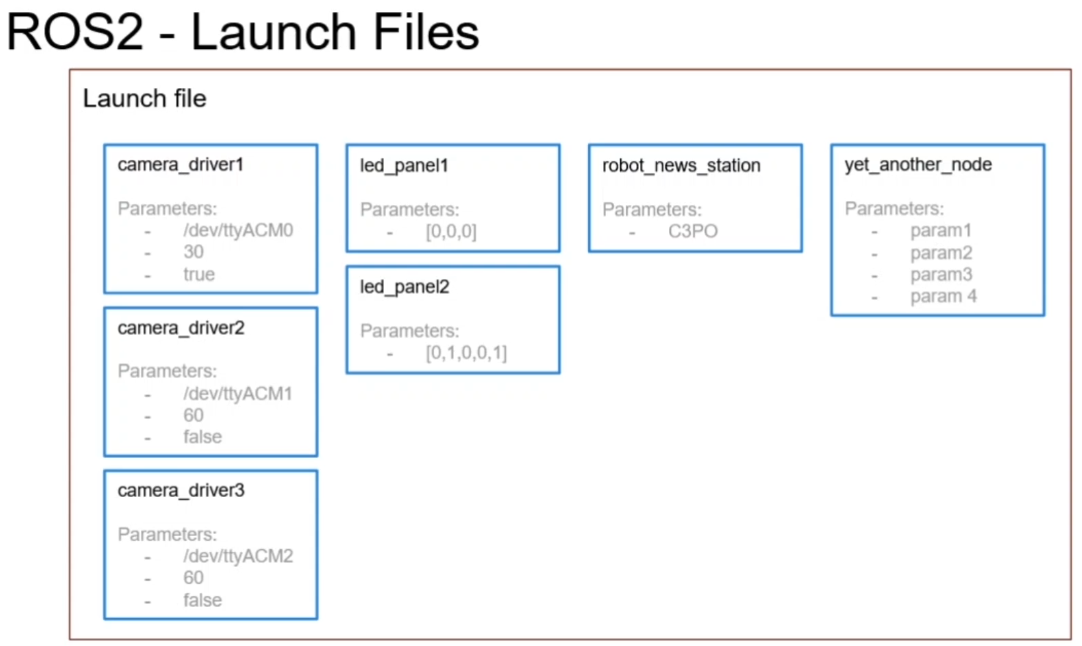


1. Open the package folder with vscode
2. Get inside the script file, press ctrl+shift+p
3. In search bar type C/C++ → choose “C/C++ : Edit configuration(JSON)
4. This will pop up c\_cpp\_properties.json file
5. Inside “includePath” add opt folder ros-dashing include folder



Launch file:

The launch file allows launching multiple nodes in a single file with associated params.



Writing launch files, it is always better to create a separate new package for keeping all the launch files so that it avoids the conflict of dependencies with base packages.

Create a launch file package

$ ros2 pkg create ros2\_bringup

Note: No dependencies required for creating launch package, it automatically use ament\_cmake as built type.

Create launch folder inside the package & create launch file

$ cd ros2\_bringup

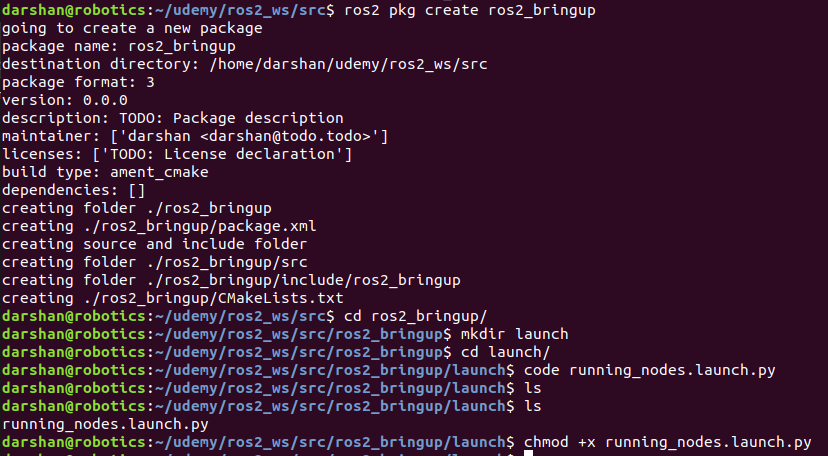
$ mkdir launch

$ cd launch

$ code running\_nodes.launch.py

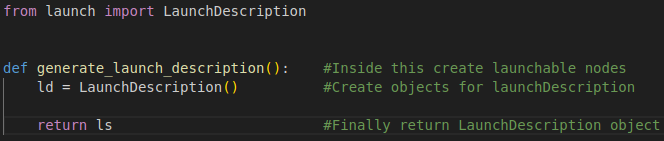
Give permission

$ chmod +x running\_nodes.launch.py

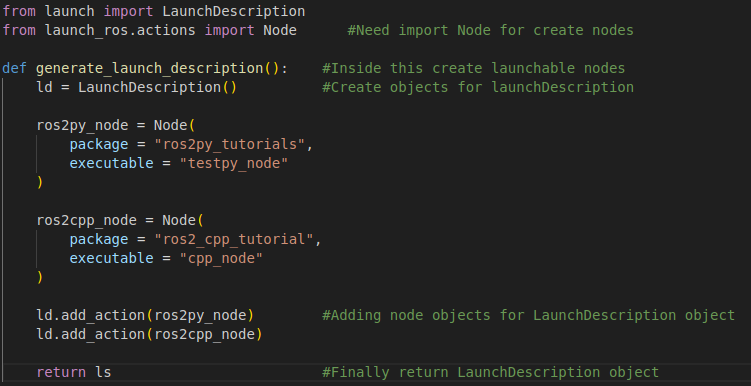


Note: Need to build this launch package with **--symlink-install.** It avoids building the package whenever new changes are added.

Minimal content inside the launch file.



Launch file with node executables

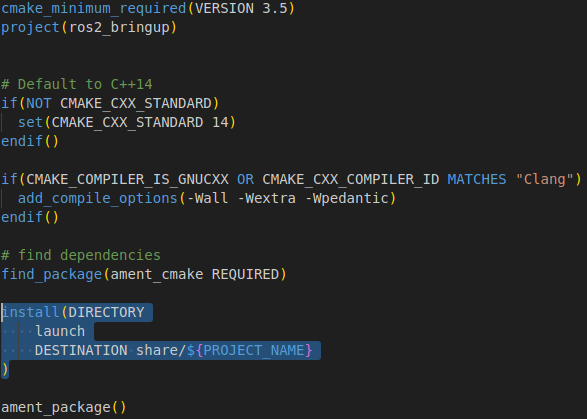


**Note**: \*\*\*If you’re using the ros2 dashing distribution, you have to replace the “**executable**” variable with “**node\_executable**” inside the node executable object.

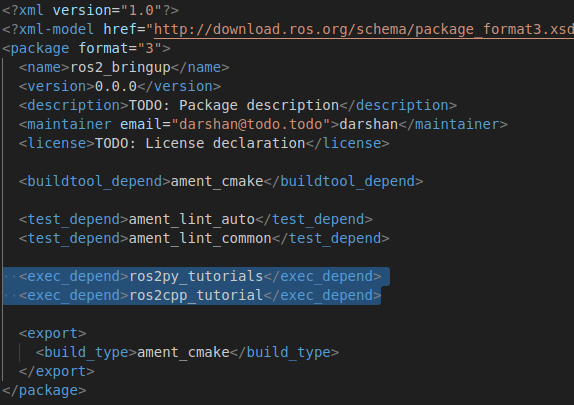
node\_executable = "cpp\_node"

**Edit CMakeList and package.xml files**

CMakeList : Adding the launch folder.



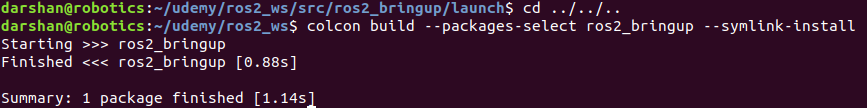
Package.xml : Adding executable nodes carrying package with <exec\_depend>.



Finally build the package using colcon build with --symlink-install .

$ cd ros2\_ws

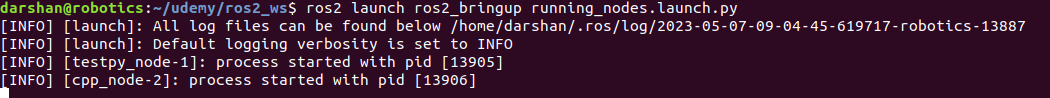
$ colcon build --packages-select ros2\_bringup --symlink-install



Running launch file

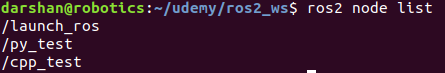
$ source install/local\_setup.bash

$ ros2 launch ros2\_bringup running\_nodes.launch.py



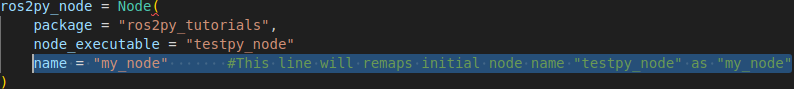
Inspecting nodes using ros2 command line tool

$ ros2 node list

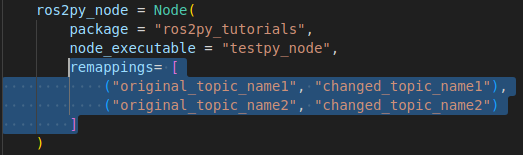


Advanced launch files.

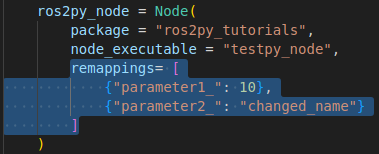
1. Remapping node name



1. Remapping topic name or services name [**Note**: Use tuple/curve brackets]

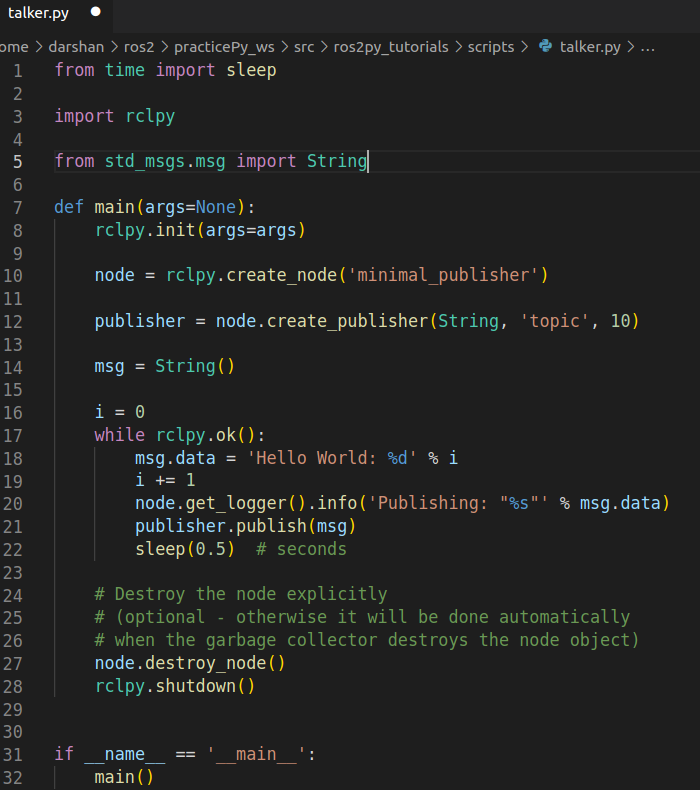


1. Remapping parameters [**Note**: Use dictionary brackets]



**Step 2:**

Create a ROS 2 node script inside the package inside ros2py\_tutorials[package\_name] folder.



**Step 3:** Edit setup.py for compilation

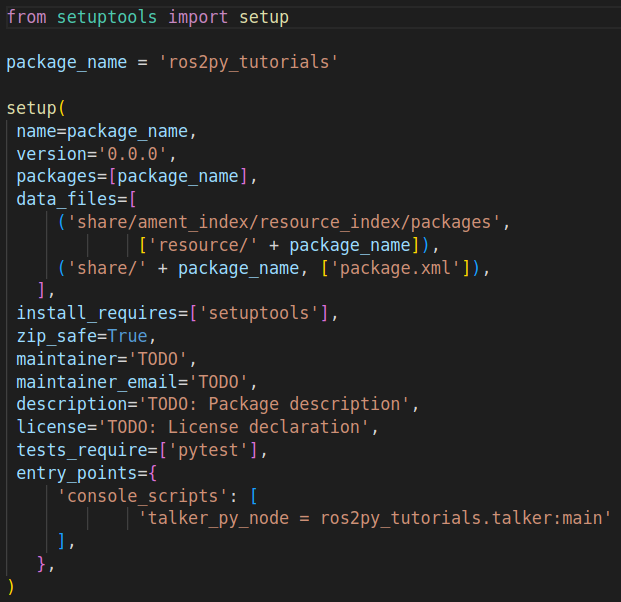
entry\_points = {‘console\_scripts’ : ['talker\_py\_node = ros2py\_tutorials.talker:main'] }

Where ,

talker\_py\_node → is the name of executable

ros2py\_tutorials.talker:main → execute the main function of talker

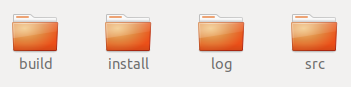
Ex:



**Step 4:** Build the package

colcon build --symlink-install

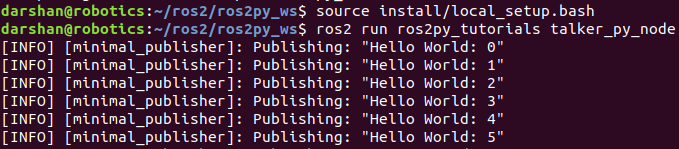
After successful compilation you will see build install log folders along with src



**Step 5:** Source and Run node

source install/local\_setup.bash

ros2 run ros2py\_tutorials talker\_py\_node



Write a subscriber node and follow the similar steps to compile and execute it.

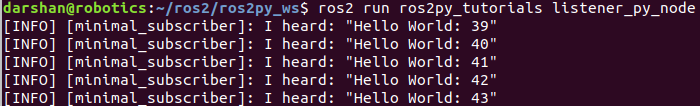
Like,

Place the subscriber node name called listener\_node inside the package inside ros2py\_tutorials folder.

Add 'listener\_py\_node = ros2py\_tutorials.listener:main' inside setup.py

Compile using colcon build.

Run node: ros2 run ros2py\_tutorials listener\_py\_node

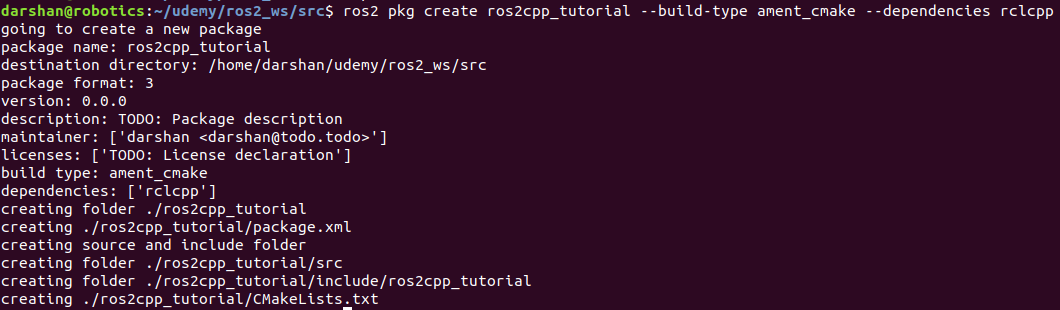


Launch file creation:

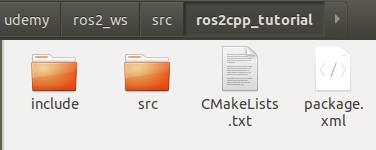
**ROS 2 C++ package creation**

Command for creating cpp package with build tool and dependencies.

$ ros2 pkg create ros2cpp\_tutorial --build-type ament\_cmake --dependencies rclcpp

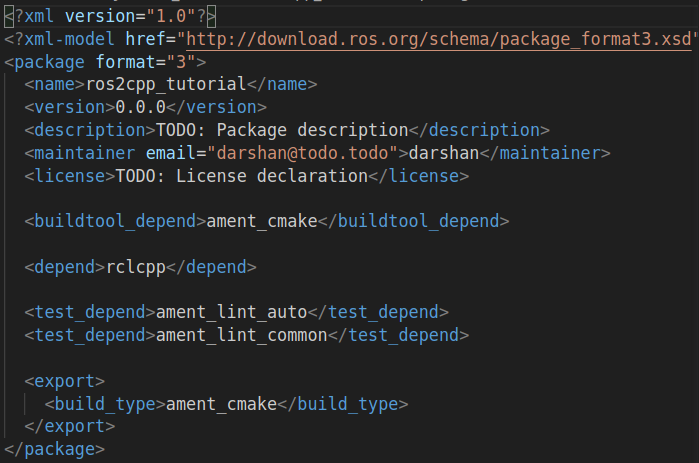


Architecture of c++ package



Keep all cpp nodes in src folder

Package.xml file



**Note:** Since compiling cpp scripts, build\_type should be **“ament\_cmake”.**

<build\_type>ament\_cmake</build\_type>

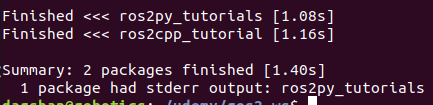
<depend>rclcpp</depend>

Then it has a traditional CMakeList.txt file similar to ROS1 to support for the compiling cpp files.

Build package

$ cd ~ros2\_ws

$ colcon build



**Building particular package:**

$ colcon build --packages-select ros2cpp\_tutorials

Note: ROS2 Installation key-error problem solving link

<https://stackoverflow.com/questions/74794022/curl-28-failed-to-connect-to-raw-githubusercontent-com-port-443-connection-t>

**Creating custom messages**

ROS 2 provides various inbuilt standard message formats to directly work with your application.

Sometimes you might need a specific message format relevant to your particular use case, that time ROS 2 will allow you to easily create your data types based message format.

Below steps describe the procedure in detail on creating your own custom message.

1. Create an interface package.
2. Create an msg folder inside the package.
3. Create a .msg file inside the msg folder and data types
4. Edit CMakeList & package.xml file
5. Compile the package

**Create an interface package**

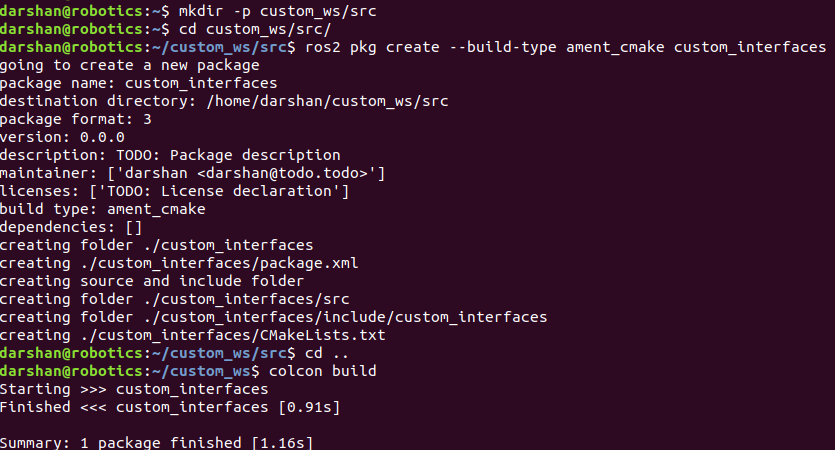
mkdir -p custom\_ws/src

cd custom\_ws/src

ros2 pkg create --build-type ament\_cmake custom\_interfaces

cd ..

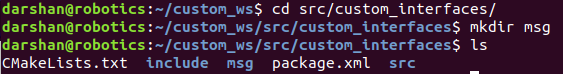
colcon build



**Create msg directory inside package**

cd custom\_ws/src/custom\_interfaces

mkdir msg



**Create a .msg file inside the msg folder.**

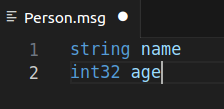
cd custom\_ws/src/custom\_interfaces/msg

code Person.msg



Note: \*\*The file name should start with Capital letter. Ex: Person.msg or Test.msg.

Add data types inside the Person.msg file



Note: \*\*The message data types should start with lowercase letters. Ex: string name

**Edit CMakeList & package.xml file**

**Add the below information inside CmakeList file before ament\_cmake():**

find\_package(rosidl\_default\_generators REQUIRED)

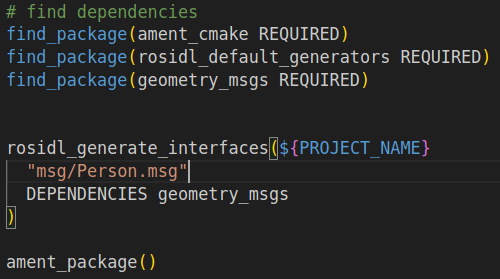
find\_package(geometry\_msgs REQUIRED)

rosidl\_generate\_interfaces(${PROJECT\_NAME}

"msg/Person.msg"

DEPENDENCIES geometry\_msgs

)



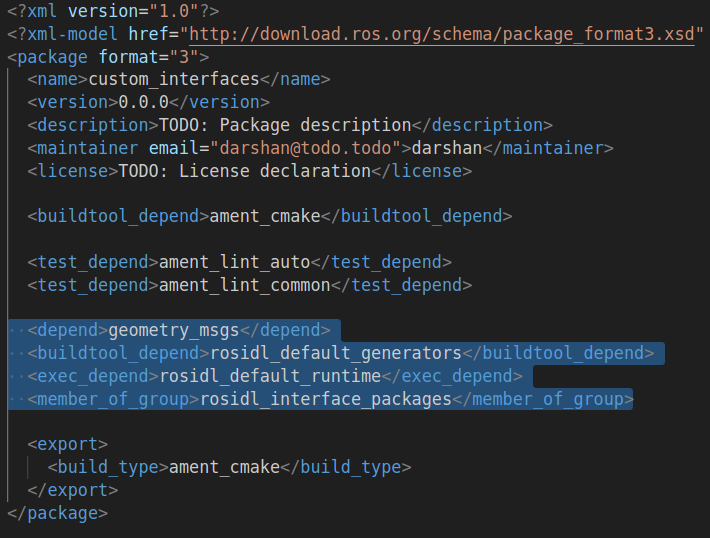
**Add the below information inside Package.xml file**

<depend>geometry\_msgs</depend>

<buildtool\_depend>rosidl\_default\_generators</buildtool\_depend>

<exec\_depend>rosidl\_default\_runtime</exec\_depend>

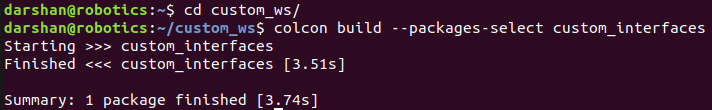
<member\_of\_group>rosidl\_interface\_packages</member\_of\_group>



**Finally compile the package**

cd cust om\_ws

colcon build --packages-select custom\_interfaces



Using custom message formats in publisher and subscriber scripts. Have to create new package in order to use custom messages

Now you’re ready to use this custom message inside your publisher and subscriber scripts. So you can create a new package and inside that write a publisher and subscriber script using this custom message.

Creating new package

cd ~custom\_ws/src

ros2 pkg create --build-type ament\_cmake custom\_pkg

cd ..

colcon build

Get into the src folder and write publisher and subscriber code using Person.msg message format.

cd ~custom\_ws/src/custom\_pkg/src

code publisher.cpp

#include <chrono>

#include <memory>

#include "rclcpp/rclcpp.hpp"

#include "custom\_interfaces/msg/person.hpp" // CHANGE

using namespace std::chrono\_literals;

class MinimalPublisher : public rclcpp::Node

{

public:

MinimalPublisher()

: Node("minimal\_publisher"), count\_(0)

{

publisher\_ = this->create\_publisher<custom\_interfaces::msg::Person>("topic", 3); // CHANGE

timer\_ = this->create\_wall\_timer(

500ms, std::bind(&MinimalPublisher::timer\_callback, this));

}

private:

void timer\_callback()

{

auto message = custom\_interfaces::msg::Person();

message.name = "Darshan"; // CHANGE

message.age = 26; // CHANGE

RCLCPP\_INFO(this->get\_logger(), "I am sending : '%s' '%d'", message.name.c\_str(), message.age);

publisher\_->publish(message);

}

rclcpp::TimerBase::SharedPtr timer\_;

rclcpp::Publisher<custom\_interfaces::msg::Person>::SharedPtr publisher\_; // CHANGE

size\_t count\_;

};

int main(int argc, char \* argv[])

{

rclcpp::init(argc, argv);

rclcpp::spin(std::make\_shared<MinimalPublisher>());

rclcpp::shutdown();

return 0;

}

code subscriber.cpp

#include <memory>

#include "rclcpp/rclcpp.hpp"

#include "custom\_interfaces/msg/person.hpp"

using std::placeholders::\_1;

class MinimalSubscriber : public rclcpp::Node

{

public:

MinimalSubscriber()

: Node("minimal\_subscriber")

{

subscription\_ = this->create\_subscription<custom\_interfaces::msg::Person>(

"topic", 10, std::bind(&MinimalSubscriber::topic\_callback, this, \_1));

}

private:

void topic\_callback(const custom\_interfaces::msg::Person::SharedPtr msg) const

{

RCLCPP\_INFO(this->get\_logger(), "I heard: '%s' '%d'", msg->name.c\_str(), msg->age);

}

rclcpp::Subscription<custom\_interfaces::msg::Person>::SharedPtr subscription\_;

};

int main(int argc, char \* argv[])

{

rclcpp::init(argc, argv);

rclcpp::spin(std::make\_shared<MinimalSubscriber>());

rclcpp::shutdown();

return 0;

}

Edit CmakeLists.txt

Add executables and target dependencies

find\_package(rclcpp REQUIRED)

add\_executable(talker src/publisher.cpp)

ament\_target\_dependencies(talker rclcpp custom\_interfaces) # CHANGE

add\_executable(listener src/subscriber.cpp)

ament\_target\_dependencies(listener rclcpp custom\_interfaces) # CHANGE

install(TARGETS

talker

listener

DESTINATION lib/${PROJECT\_NAME})

Edit Package.xml

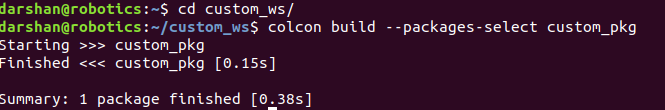
Add “custom\_interfaces” depend block

<depend>custom\_interfaces</depend>

Now compile the package

cd ~custom\_ws

colcon build --packages-select custom\_pkg



Custom service

Below steps describe the procedure in detail on creating your own custom message.

Create an srv folder inside the package.

Create a .srv file inside the msg folder and data types

Edit CMakeList & package.xml file

Compile the package

Create an srv folder inside the package.

cd custom\_ws/src/custom\_interfaces

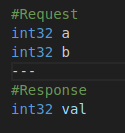
mkdir srv



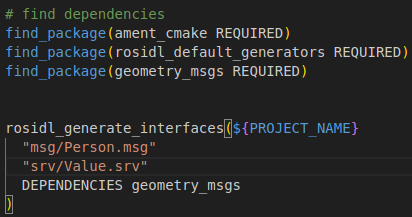
Create a .srv file inside the msg folder and data types

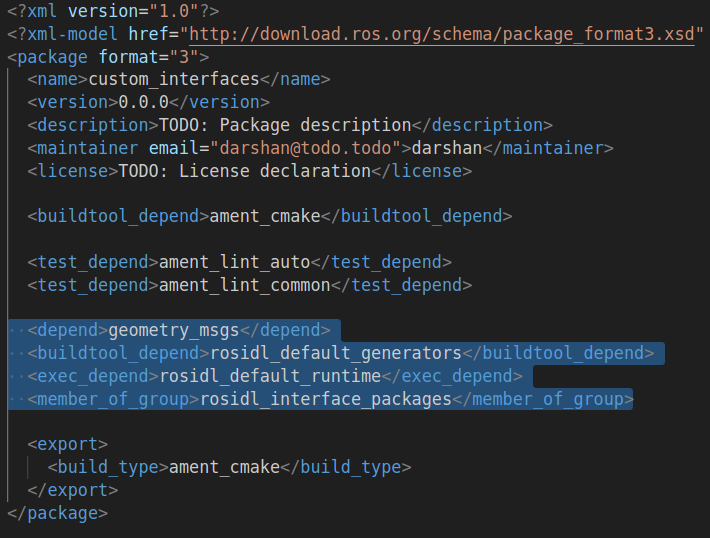
cd custom\_ws/src/custom\_interfaces/srv

code Value.srv



Edit CMakeLists.txt



Package.xml  


Compile package

cd custom\_ws

colcon build --packages-select custom\_interfaces

Note: While compiling package sometime you’ll get below quoted error

“ rosidl\_adapter.parser.InvalidServiceSpecification: Could not find separator

'---' between request and response “



To avoid this error, completely remove separation line ie, - - - inside .srv file rewrite “---”

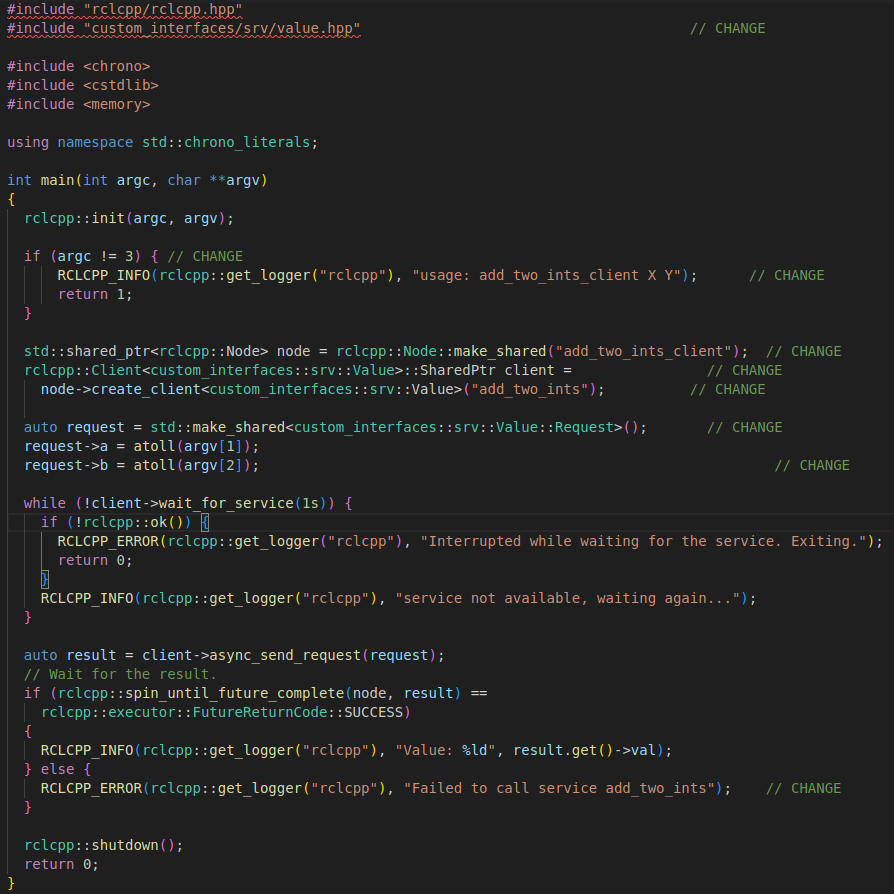
Writing service server and service client script using this custom Value.srv file.

For that get into custom\_pkg folder and write server & client script inside the src folder

service\_server.cpp



service\_client.cpp



Edit CmakeLists.txt

#Custom service

add\_executable(server src/service\_server.cpp)

ament\_target\_dependencies(server

rclcpp custom\_interfaces) # CHANGE

add\_executable(client src/service\_client.cpp)

ament\_target\_dependencies(client

rclcpp custom\_interfaces) # CHANGE

install(TARGETS

server

client

DESTINATION lib/${PROJECT\_NAME})

Editit Package.xml

<depend>custom\_interfaces</depend>

Commands

**## Custom service**

Terminal 1:

ros2 run custom\_pkg server

Terminal 2:

ros2 run custom\_pkg client 10 9

**ROS TOOLS:**

1. Remapping the ROS 2 node while executing on command line

**$ ros2 run <package\_name> <node\_name> --ros-args --remap \_\_node:=<remapping\_name>**

**$ ros2 run training talker --ros-args --remap \_\_node:=abc**

**(or)**

**$ ros2 run training talker --ros-args -r \_\_node:=abc**

Note: By using this concept you can run the same node content with multiple times by changing node name.

EX:

**$ ros2 run training talker --ros-args --remap \_\_node:=node1**

**$ ros2 run training talker --ros-args --remap \_\_node:=node2**

**$ ros2 run training talker --ros-args --remap \_\_node:=node3**

1. Building python packages : In ROS2 whenever you have made some changes in ros2.py code you need to compile the code again using colcon build options similar to ros1& ros2 cpp node. In order to avoid this you can use --symlink-install option at the time of building your ros2 python package; it will avoid building python script again and again after making changes in code.

**$ colcon build --packages-select action\_tutorials\_py --symlink-install**

4. ROS 2 topic debug and remapping

Allow ros2 publisher to publish at specific hz[ex: 10 hz]

$ ros2 topic pub -r 10 /talker <message\_type> <message\_data>

Find Bandwidth of topic

$ ros2 topic bw /talker

Remapping topic at runtime

**$ ros2 run <package\_name> <node\_name> --ros-args --remap <original\_topic>:=<remapping\_topic\_name>**

**$ ros2 run training talker --ros-args --remap /topic:=/abc**

**(or)**

**$ ros2 run training talker --ros-args -r /topic:=/abc**