From the course : https://www.udemy.com/course/ros-basics-program-robots/

Notes by: Anirudh.P

To create a new package :

1. mkdir ‘workspace name’
2. In the ws, mkdir src
3. In src, catkin_init_workspace
4. Go to root folder of your workspace and catkin_make
5. Go to src, and catkin_create_pkg robot_tutorials rospy roscpp std_msgs (the dependencies of your package)
6. Go to root directory of your workspace again and catkin_make (because the package you created, has to be built)

How to add executable files :

1. In src create your file (let’s take a .cpp file)
2. Then go to CMakeLists.txt and scroll down to #add_executable(walle_hello_node src/hello-eva.cpp), and remove the # and modify it to this add_executable(name_of_node src/nameofprogram.cpp)
3. Then go to root folder of your workspace and catkin_make (now your executable will be built)
4. Now under the root folder of your workspace go to devel/lib/name_of_your_package and here you will find your executable. If you have sourced the setup.bash in this workspace then you can run the executable from anywhere using rosrun package_name executable_name

roscore - to run the rosmaster

Installation directory of ROS - cd /opt/ros/melodic/bin/

If you want to change the name of a ros node running - rosrun package_name node_name __name:=newname

Turtlesim :

rosrun turtlesim turtlesim_node
rosrun turtlesim turtle_teleop_key

rostopic commands :

```
rostopic -h (describes all the arguments you can provide to the command rostopic

rostopic list
rostopic echo /topic_name
rostopic type /turtle1/cmd_vel (shows the message type)
Output : geometry_msgs/Twist

rosmsg show geometry_msgs/Twist (gives a details of the message type)

rosrun rqt_graph rqt_graph (shows the graphical representation of all the nodes and how the
communication is between them)

rostopic info /name_of_topic (gives message type of topic and shows the node that is publishing
to the topic and the node that is subscribing to the same topic)


rostopic pub -1 /turtle1/cmd_vel geometry_msgs/Twis[1.0, 0.0, 0.0]' '[0.0, 0.0, 4.0]' (command to
publish once to a topic)

rostopic hz /turtle1/cmd_vel (shows the frequency at which a rostopic is being published)
```

Publisher Code :

```
#!/usr/bin/env python3
#This is required for when you use rosrun to execute this node from anywhere, it will tell ROS
what to use to run this program

import rospy
from std_msgs.msg import String, Int8

def function_to_publish():
        hello_pub = rospy.Publisher('Topic_Name', String, queue_size=10)
        hello_pub_int = rospy.Publisher('Int8_type_topic', Int8, queue_size=10)
        rospy.init_node('Node_Name', anonymous = True)
        rate = rospy.Rate(10)
        counter = 0
        while not rospy.is_shutdown():
                greeting = "Hello, This is the message that is being broadcasted"
                greeting_int = ord(greeting[counter%len(greeting)])
                rospy.loginfo(greeting) #For Debugging (it prints our published message on the
terminal and writes it to the nodes logfile and rosout)
```

```
        hello_pub.publish(greeting) #The command that publishes the message
        hello_pub_int.publish(greeting_int) #This command also publishes, but to the
topic 'Int8_type_topic' and with an 8bit Integer type
        counter += 1
        rate.sleep() #Makes it sleep enough for publishing once every 100 milliseconds



if __name__ == '__main__':
    try:
        function_to_publish()
    except rospy.ROSInterruptException:
        pass
```

Subscriber Node :

```
#!/usr/bin/env python3
#This is required for when you use rosrun to execute this node from anywhere, it will tell ROS
what to use to run this program

import rospy
from std_msgs.msg import String, Int8

def callback_str(data):
    rospy.loginfo(data.data)

def callback_int(data):
    rospy.loginfo(str(data.data))

def Eva_listener():
    rospy.init_node('Subscribing_Node', anonymous = False)
    rospy.Subscriber('Topic_Name', String, callback_str) #callback_str here is the name of
the function that is to be called when the message is received from the topic 'Topic_Name'
    rospy.Subscriber('Int8_type_topic', Int8, callback_int)
    rospy.spin() #Keeps the node from exiting, until the node has been shutdown!

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

ROS Service and Parameters :

Services : rosservices accept inputs as requests in a predefined format and respond according to their functional implementation

A node hosts services

In the case of turtle sim node it hosts a few services, try in terminals:

1. roscore
2. rosrun turtlesim turtlesim_node
3. rosrun turtlesim turtle_teleop_key

To check all the services a node is offering type :

4. rosnode list
5. rosnode info /turtlesim
6. rosservice list - to check the available services in the graph at any moment
7. rosservice -h - can be used to find the usage of various ros service commands
8. rosservice info /servicename - to get detailed info about the running service
   a. rosservice info /reset - this will have a service of type empty, which means that it takes no arguments
9. rosservice call /resets - calls the service and this service completely re instantiates the 2d turtle robot
10. rosservice call /clear - erases any history of the path taken by the robot
11. rosservice info /spawn 7 2 3.14 “”- under type will show you that this service takes 3 arguments, which specifies the pose and the coordinates that the robot should take while creating a new instance of the turtle
12. rostopic list - this will show you a new set of topics under turtle2

Parameters : Allow you to store and manipulate data of any format

1. rosparam -h
2. rosparam list - to view the parameters currently hosted by the ros parameter’s server
3. rosparam get /background_b - outputs the value that a parameter currently holds
4. rosparam set /parameter_name 0 - allows you to set the value of a parameter
   a. rosparam set /background_b 0
   b. rosparam set /background_g 0
   c. rosparam set /background_r 0
   d. These 3 commands will change the value of these parameters to 0, in other words this will now change the colour of the turtle sim canvas to black
5. rosservice call /clear - to make the above parameters changes take effect

Parameter servers are used to hold constants, for example you can have your own parameters server and this data can be used by any node in the graph to fetch and use.

ROS Message and Service files :

Message files : Holds the fields of a ROS message type.

Service files : Describes the content of a service. It has a request field and a response field.
To create a message file :

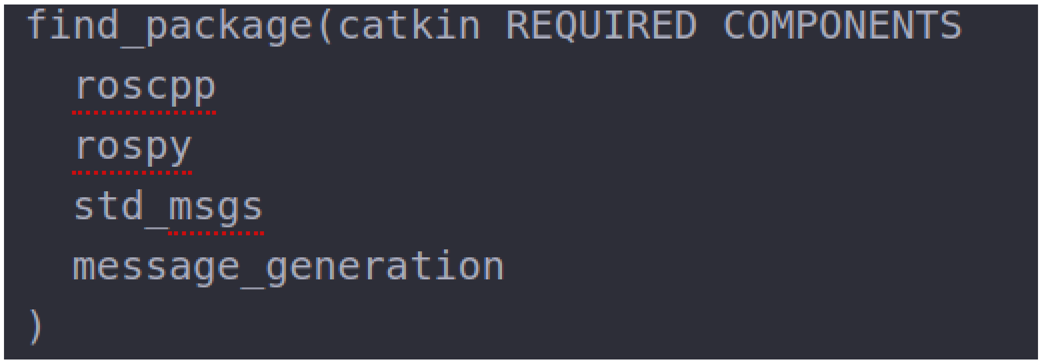
1. Go to your package, create a folder for messages, create a new file with the extension ".msg"
   a. Sample message type with fields for robots name, country and product Id :
      i. string robot_name
         string country
         uint8 product_id

Similarly, to create a service file :

1. Go to your package, create a folder for services, create a new file with the extension ".srv"
   a. Sample service which takes two integers as input and returns the maximum among the two. For this the service file shall contain 2 integers in the request and 1 integer in the response (the type of these request/response fields has to be one that belongs to the standard message types). In this sample we will be using the type 'int64':
      i. int64 a
         int64 b
         ---
         int64 max

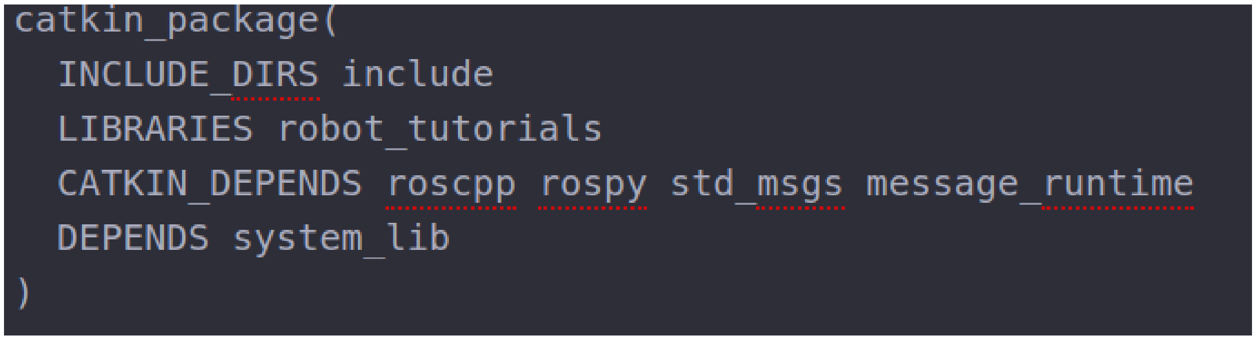
To build the message and service files for usage, we need to add a few fields in our CMakeLists.txt file:

1. In the find_package call add the dependency : "message_generation"
   a. Like this :



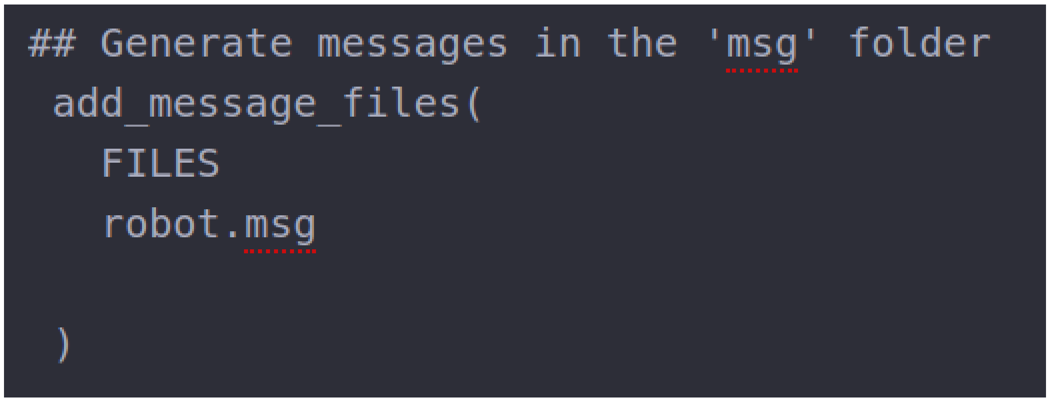
```
find_package(catkin REQUIRED COMPONENTS
  roscpp
  rospy
  std_msgs
  message_generation
)
```

2. Search for the generate messages tag and uncomment it
3. In the catkin package call, uncomment it and add the dependency “message_runtime” in the line catkin_depends. Like this :



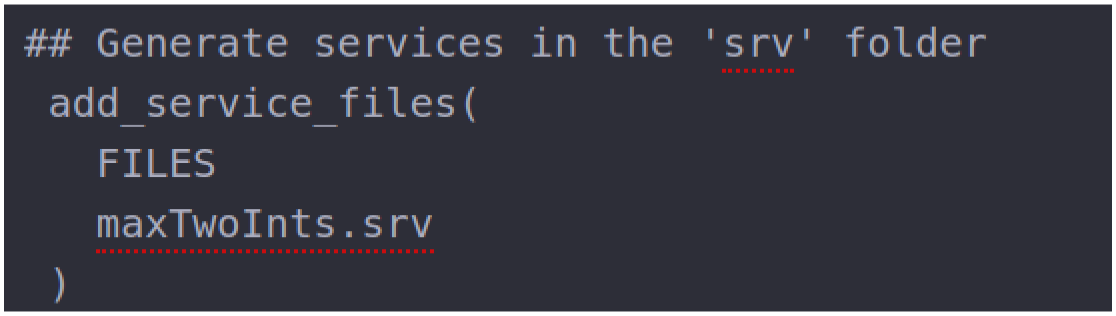
```
catkin_package(
  INCLUDE_DIRS include
  LIBRARIES robot_tutorials
  CATKIN_DEPENDS roscpp rospy std_msgs message_runtime
  DEPENDS system_lib
)
```

4. In the add_message_files call, uncomment the lines, remove the sample message files and add the name of your message file. Like this:



```
## Generate messages in the 'msg' folder
 add_message_files(
   FILES
   robot.msg

 )
```

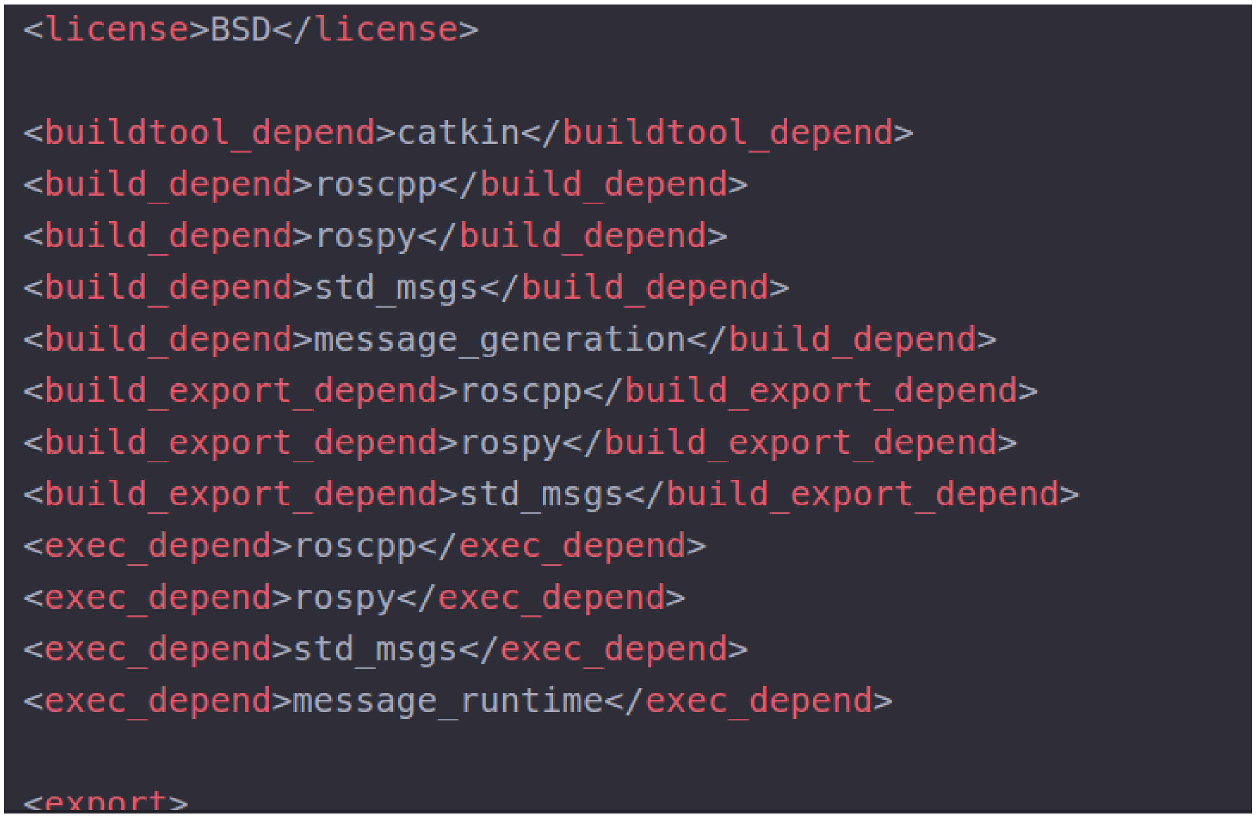
5. Similarly, in the add_service_files call, uncomment the lines, remove the sample service files and add the name of your service file. Like this:



```
## Generate services in the 'srv' folder
 add_service_files(
   FILES
   maxTwoInts.srv
 )
```

6. Now open your package.xml file and add a new build depend tag for message generation : “<build depend>message_generation</build_depend>. And then add an execution depend tag for message runtime : “<exec_depend>message_runtime</exec_depend>. We add these so that the message_generation will be used during build time and message_runtime during

execution time. Like this:



```
<license>BSD</license>

<buildtool_depend>catkin</buildtool_depend>
<build_depend>roscpp</build_depend>
<build_depend>rospy</build_depend>
<build_depend>std_msgs</build_depend>
<build_depend>message_generation</build_depend>
<build_export_depend>roscpp</build_export_depend>
<build_export_depend>rospy</build_export_depend>
<build_export_depend>std_msgs</build_export_depend>
<exec_depend>roscpp</exec_depend>
<exec_depend>rospy</exec_depend>
<exec_depend>std_msgs</exec_depend>
<exec_depend>message_runtime</exec_depend>

<export>
```

7. Then go to the initial workspace directory in which your package is in and run “catkin_make install”, to build and install code for use in all supported languages.
8. To ensure proper installation, we can run the command “rosmsg show package_name/message_name” (it will list its content) and similarly for service files “rossrv show package_name/service_name.

Services and Client Code :

Here we are writing code for a rosservice that accepts 2 integers and returns the maximum of the two (Both server and client). The client will be able to send and receive requests/responses from the maxTwoInts server in the formats specified in your service file.

1. Go to your package, then scripts folder and create a file called ‘max_two_ints_server.py’
   SERVER CODE:

```
#!/usr/bin/env python

from robot_tutorials.srv import *
import rospy

def handle_max_two_ints(req):
    # Return the max of the two integers
    print("Returning max(%s, %s) = %s"%(req.a, req.b, max(req.a, req.b)))
    return maxTwoIntsResponse(max(req.a, req.b))

def max_two_ints_server():
    # rospy.init_node this will register the node on the graph
    rospy.init_node('max_two_ints_server')
    # rospy.service takes 3 arguments: the name of the service, the type of the
service and the callback function
    m = rospy.Service('max_two_ints', maxTwoInts, handle_max_two_ints)
    print("Ready to compute max(a,b)")
    # rospy.spin() keeps your code from exiting until the service is shutdown
    rospy.spin()

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

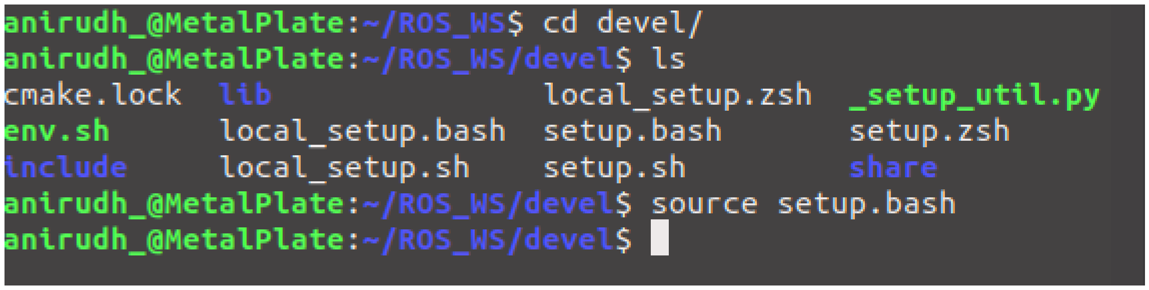
2. Now create a file called ‘max_two_ints_client.py’

   CLIENT CODE:

```
#!/usr/bin/env python

import sys
import rospy
from robot_tutorials.srv import *

def max_two_ints_client(x, y):
    # The wait_for_service blocks further execution of the function until a service
under the name "max_two_ints" is available
    rospy.wait_for_service("max_two_ints")
    try:
        # Creates a handle for our request and then we can call it with the two
integers as its arguments
        max_two_ints = rospy.ServiceProxy('max_two_ints', maxTwoInts)
        resp = max_two_ints(x, y)
        return resp.max
    # If the call fails, a service exception is thrown, which is caught here with an
except block
    except rospy.ServiceException, e:
        print("Service call failed: %s"%e)

# This is to let the user know the proper usage of the client function
def usage():
    return("%s [%s %s]"%sys.srgv[0])
```

```
if __name__ == '__main__':
        # We'll receive two values from the input and pass them along to the client
function
        if len(sys.argv) == 3:
                x = int(sys.argv[1])
                y = int(sys.argv[2])
        else:
                print usage()
                sys.exit(1)
        print("Requesting max(%s, %s)"%(x, y))
        print("max(%s, %s) = %s"%(x, y, max_two_ints_client(x, y)))
```

3. Now let's test if the codes work properly. Open a new terminal and start the rosmaster by running "roscore" in your terminal. In another terminal, go to your workspace and source the setup.bash. Like this :



```
anirudh_@MetalPlate:~/ROS_WS$ cd devel/
anirudh_@MetalPlate:~/ROS_WS/devel$ ls
cmake.lock  lib                local_setup.zsh  _setup_util.py
env.sh      local_setup.bash   setup.bash       setup.zsh
include     local_setup.sh     setup.sh         share
anirudh_@MetalPlate:~/ROS_WS/devel$ source setup.bash
anirudh_@MetalPlate:~/ROS_WS/devel$ 
```

4. Then go back to your workspace and "catkin_make"
5. Then go to your folder where the server and client scripts are and change theme into executable by changing their permissions "chmod +x max_two_ints_*"
6. After that you are now ready to run the server and client files. To start our server, in a new terminal use "rosrun package_name service_codefilename"
    a. You can check if our new service is registered by opening a new terminal and running "rosservice list"
    b. You can also run "rosservice info /nameofyourservice" and you will be able to see the type and the argument the service takes
7. Next open another terminal (remember to source your workspace) and here if we run the client code with 2 integer arguments ""rosrun package_name client_codefilename argument1 argument2" and we will see the debug messages for the server and client are printed.
8. Alternatively, we can also call the service using "rosservice call /service_name argument1 argument2". Here you will only get the expected results.

ROSBAGS :

Rosbags are used to record data in ROS and play it back in a later time to replicate similar behaviour.

1. In separate terminals run the rosmaster “roscore”, the turtle sim node “rosrun turtlesim turtlesim_node”, the turtle teleop key “rosrun turtlesim turtle_teleop_key”.
2. Then in another terminal create a new directory to store the recorded data, navigate to it and run “rosbag record -a” to start capturing all data into a bag file.
3. Now move the turtle around for sometime and the rosbag in the background will capture all the data. After that you can stop the rosbag recording.
4. In the same terminal type “rosbag info rosbagname.bag” to display the detailed information about the bagfile’s recording.
5. And now to play the data back, simply run “rosbag play rosbagname.bag” and you will see the turtle simulating exactly how you moved it from the start till the end.

Rosbags will be useful to record, for example real time sensor and actuator information and then use them later in simulation or data analytics.