

---

## **ROS Basics in 5 days**

---

### **Unit 6 Services in ROS: Servers & Messages**



- Summary -

**Estimated time to completion:** 3 hours

**What will you learn with this unit?**

- [How to give a service](#)
- [Create your own service server message](#)

- End of Summary -

## 6.1 Service Server

Until now, you have just been calling services that other nodes provided. But now, you are going to create your own service!

- Example 6.7 -

First, create the package **my\_service\_server\_example\_pkg**, executing the following commands in the Web Shell:

► Execute in Shell #1

```
In [ ]: cd ~/catkin_ws/src
```

```
In [ ]: catkin_create_pkg my_service_server_example_pkg rospy std_msgs
```

Now create the **simple\_service\_client.py** script inside it:

► Execute in Shell #1

```
In [ ]: cd ~/catkin_ws/src/my_service_server_example_pkg
```

```
In [ ]: mkdir scripts
```

```
In [ ]: cd scripts
```



```
In [ ]: touch simple_service_server.py
```



```
In [ ]: chmod +x simple_service_server.py
```



#### Python Program: simple\_service\_server.py

```
In [ ]: #!/usr/bin/env python

import rospy
from std_srvs.srv import Empty, EmptyResponse # you import the service message python classes generated from

def my_callback(request):
    print("My_callback has been called")
    return EmptyResponse() # the service Response class, in this case EmptyResponse
    #return MyServiceResponse(Len(request.words.split()))

rospy.init_node('service_server')
my_service = rospy.Service('/my_service', Empty, my_callback) # create the Service called my_service with the
rospy.spin() # maintain the service open.
```



#### END Python Program: simple\_service\_server.py

And now execute it in a Terminal:

► Execute in Shell #1

```
In [ ]: rosrn my_service_server_example_pkg simple_service_server.py
```



Did something happen?

Of course not! At the moment, you have just created and started the Service Server. So basically, you have made this service available for anyone to call it.

This means that if you do a **rosservice list**, you will be able to visualize this service on the list of available services.

► Execute in Shell #2

```
In [ ]: rosservice list
```



On the list of all available services, you should see the **/my\_service** service.

Shell #2 Output

```
/base_controller/command_select  
/bb8/camera1/image_raw/compressed/set_parameters  
/bb8/camera1/image_raw/compressedDepth/set_parameters  
/bb8/camera1/image_raw/theora/set_parameters  
...  
/my_service  
...
```

Now, you have to actually **CALL** it. So, call the **/my\_service** service manually. Remember the calling structure discussed in the previous chapter and don't forget to TAB-TAB to autocomplete the structure of the Service message.

► Execute in Shell #2

```
In [ ]: rosservice call /my_service [TAB]+[TAB]
```



Did it work? You should've seen the message **'My callback has been called'** printed at the output of the the Python code. Great!

```
user:~/catkin_ws/src/my_examples_pkg/scripts$ rosrn my_examples_pkg simple_service_server.py
My_callback has been called
[]
```

**INFO:** Note that, in the example, there is a commented line in the **my\_callback** function. This gives you an example of how you would access the request given by the caller of your service. It's always **request.variables\_in\_the\_request\_part\_of\_srv\_message**.

So, for instance, let's do a flashback to the previous chapter. Do you remember Example 5.5, where you had to perform calls to a service in order to execute a certain trajectory with the robotic arm? Well, for that case, you were passing the name of the trajectory to execute to the Service Server in a variable called **traj\_name**. So, if you wanted to access the value of that **traj\_name** variable in the Service Server, you would have to do it like this:

```
In [ ]: request.traj_name
```



Quite simple, right?

That commented line also shows you what type of object you should return. Normally, the **Response** Python class is used. It always has the structure **name\_of\_the\_messageResponse()**. That's why for the example code shown above, since it uses the **Empty** service message, the type of object that returns is **EmptyResponse()**. But, if your service uses another type of message, let's say one that is called **MyServiceMessage**, then the type of object that you would return would be **MyServiceMessageResponse()**.

- End of Example 6.7 -

- Exercise 6.2 -

- The objective of Exercise 6.2 is to create a service that, when called, will make BB8 robot move in a circle-like trajectory.
- You can work on a new package or use the one you created for Exercise 5.1, called **unit\_5\_services**.

- Create a Service Server that accepts an **Empty** service message and activates the circle movement. This service could be called **/move\_bb8\_in\_circle**.

You will place the necessary code into a new Python file named **bb8\_move\_in\_circle\_service\_server.py**. You can use the Python file [simple\\_service\\_server.py](#) as an example.

- Create a launch file called **start\_bb8\_move\_in\_circle\_service\_server.launch**. Inside it, you have to start a node that launches the **bb8\_move\_in\_circle\_service\_server.py** file.
- Launch **start\_bb8\_move\_in\_circle\_service\_server.launch** and check that, when called through the Terminal, BB-8 moves in a circle.
- Now, create a new Python file called **bb8\_move\_in\_circle\_service\_client.py** that calls the service **/move\_bb8\_in\_circle**. Remember how it was done in the previous chapter: **Services Part 1**.

Then, generate a new launch file called **call\_bb8\_move\_in\_circle\_service\_server.launch** that executes the code in the **bb8\_move\_in\_circle\_service\_client.py** file.

- Finally, when you launch this **call\_bb8\_move\_in\_circle\_service\_server.launch** file, BB-8 should move in a circle.

- End of Exercise 6.2 -

## 6.2 How to create your own service message

So, what if none of the service messages that are available in ROS fit your needs? Then, you create your own message, as you did with the Topic messages.

In order to create a service message, you will have to follow the next steps:

- Example 6.8 -

1) Create a package like this:

► Execute in Shell #1

```
In [ ]: roscd
```

```
In [ ]: cd ..
```

```
In [ ]: cd src
```

```
In [ ]: catkin_create_pkg my_custom_srv_msg_pkg rospy
```

2) Create your own Service message with the following structure. You can put as many variables as you need, of any type supported by ROS: [ROS Message Types \(http://wiki.ros.org/msg\)](http://wiki.ros.org/msg). Create a **srv** folder inside your package, as you did with the topics **msg** folder. Then, inside this **srv** folder, create a file called **MyCustomServiceMessage.srv**. You can create with the IDE or the Terminal, as you wish.

► Execute in Shell #1

```
In [ ]: roscd my_custom_srv_msg_pkg/
```

```
In [ ]: mkdir srv
```

```
In [ ]: vim srv/MyCustomServiceMessage.srv
```

You can also create the **MyCustomServiceMessage.srv** through the IDE, if you don't feel comfortable with vim.

The **MyCustomServiceMessage.srv** could be something like this:



```
In [ ]: int32 duration    # The time (in seconds) during which BB-8 will keep moving in circles
        ---
        bool success     # Did it achieve it?
```



## 6.2.1 Prepare CMakeLists.txt and package.xml for custom Service compilation

You have to edit two files in the package similarly to how we explained for Topics:

- CMakeLists.txt
- package.xml

## 6.2.2 Modification of CMakeLists.txt

You will have to edit four functions inside CMakeLists.txt:

- [find\\_package\(\)](#)
- [add\\_service\\_files\(\)](#)
- [generate\\_messages\(\)](#)
- [catkin\\_package\(\)](#)

### I. [find\\_package\(\)](#)

All the packages needed to COMPILE the messages of topics, services, and actions go here. It's only getting its paths, and not really importing them to be used in the compilation.

The same packages you write here will go in **package.xml**, stating them as **build\_depend**.

```
In [ ]: find_package(catkin REQUIRED COMPONENTS
        std_msgs
        message_generation
        )
```



## II. add\_service\_files()

This function contains a list of all of the service messages defined in this package (defined in the srv folder).  
For our example:

```
In [ ]: add_service_files(  
        FILES  
        MyCustomServiceMessage.srv  
        )
```



## III. generate\_messages()

Here is where the packages needed for the service messages compilation are imported.

```
In [ ]: generate_messages(  
        DEPENDENCIES  
        std_msgs  
        )
```



## IV. catkin\_package()

State here all of the packages that will be needed by someone that executes something from your package. All of the packages stated here must be in the **package.xml** file as **<exec\_depend>**.


```
In [ ]: catkin_package(  
        CATKIN_DEPENDS  
        rospy  
        )
```



Once you're done, you should have something similar to this:

 CMakeLists.txt

In [ ]:



```
cmake_minimum_required(VERSION 2.8.3)
project(my_custom_srv_msg_pkg)

## Here is where all the packages needed to COMPILE the messages of topics, services and actions go.
## It's only getting its paths, and not really importing them to be used in the compilation.
## It's only for further functions in CMakeLists.txt to be able to find those packages.
## In package.xml you have to state them as build
find_package(catkin REQUIRED COMPONENTS
  std_msgs
  message_generation
)

## Generate services in the 'srv' folder
## In this function will be all the action messages of this package ( in the action folder ) to be compiled.
## You can state that it gets all the actions inside the action directory: DIRECTORY action
## Or just the action messages stated explicitly: FILES my_custom_action.action
## In your case you only need to do one of two things, as you wish.
add_service_files(
  FILES
  MyCustomServiceMessage.srv
)

## Here is where the packages needed for the action messages compilation are imported.
generate_messages(
  DEPENDENCIES
  std_msgs
)

## State here all the packages that will be needed by someone that executes something from your package.
## ALL the packages stated here must be in the package.xml as exec_depend
catkin_package(
  CATKIN_DEPENDS rospy
)
```

```
include_directories(  
  ${catkin_INCLUDE_DIRS}  
)
```

### 6.2.3 Modification of package.xml

1. Add all of the packages needed to compile the messages.

In this case, you only need to add the ***message\_generation***.

You will have to import those packages as **<build\_depend>**.

2. On the other hand, if you need a package for the execution of the programs inside your package, you will have to import those packages as **<exec\_depend>**.

In this case, you will only need to add these 3 lines to your **package.xml** file:

```
In [ ]: <build_depend>message_generation</build_depend>  
  
        <build_export_depend>message_runtime</build_export_depend>  
        <exec_depend>message_runtime</exec_depend>
```



So, at the end, you should have something similar to this:

 package.xml

In [ ]:

```
<?xml version="1.0"?>
<package format="2">
  <name>my_custom_srv_msg_pkg</name>
  <version>0.0.0</version>
  <description>The my_custom_srv_msg_pkg package</description>

  <maintainer email="user@todo.todo">user</maintainer>

  <license>TODO</license>

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

  <export>
  </export>
</package>
```



Once you're done, compile your package and source the newly generated messages:

► Execute in Shell #1

In [ ]:

```
roscd; cd ..
```



In [ ]:

```
catkin_make
```



```
In [ ]: source devel/setup.bash
```



**Important!!** When you compile new messages through `catkin_make`, there is an extra step that needs to be done. You have to type in the Terminal, in the **catkin\_ws** directory, the following command: **source devel/setup.bash**.

This command executes the bash file that sets, among other things, the newly generated messages created with **catkin\_make**. If you don't do this, it might give you a Python import error, saying that it doesn't find the message generated.

You should see among all the messages something similar to:

### ***Generating Python code from SRV my\_custom\_srv\_msg\_pkg/MyCustomServiceMessage***

To check that you have the new service message in your system, and ready to be used, type the following:

► Execute in Shell #1

```
In [ ]: rossrv list | grep MyCustomServiceMessage
```



It should output something like:

Shell #1 Output

```
my_custom_srv_msg_pkg/MyCustomServiceMessage
```

That's it! You have created your own Service Message. Now, create a Service Server that uses this type of message.

It could be something similar to this:

**Python Program:** `custom_service_server.py`

```

In [ ]: #!/usr/bin/env python

import rospy
from my_custom_srv_msg_pkg.srv import MyCustomServiceMessage, MyCustomServiceMessageResponse # you import the
                                                # generated from MyC

def my_callback(request):

    print("Request Data==> duration="+str(request.duration))
    my_response = MyCustomServiceMessageResponse()
    if request.duration > 5.0:
        my_response.success = True
    else:
        my_response.success = False
    return my_response # the service Response class, in this case MyCustomServiceMessageResponse

rospy.init_node('service_client')
my_service = rospy.Service('/my_service', MyCustomServiceMessage , my_callback) # create the Service called m
rospy.spin() # maintain the service open.

```

END Python Program: custom\_service\_server.py

- End of Example 6.8 -

- Exercise 6.3 -

- Create a new Python file called **bb8\_move\_custom\_service\_server.py**. Inside this file, modify the code you used in **Exercise 6.2**, which contained a Service Server that accepted an Empty Service message to activate the circle movement. This new service will be called **/move\_bb8\_in\_circle\_custom**. This new service will have to be called through a custom service message. The structure of this custom message is presented below:



```
In [ ]: int32 duration    # The time (in seconds) during which BB-8 will keep moving in circles
        ---
        bool success     # Did it achieve it?
```



- Use the data passed to this new **/move\_bb8\_in\_circle\_custom** to change the BB-8 behavior. During the specified **duration** time, BB-8 will keep moving in circles. Once this time has ended, BB-8 will then stop its movement and the Service Server will return a **True** value (in the **success** variable). Keep in mind that even after BB-8 stops moving, there might still be some rotation on the robot, due to inertia.
- Create a new launch file called **start\_bb8\_move\_custom\_service\_server.launch** that launches the new **bb8\_move\_custom\_service\_server.py** file.
- Test that when calling this new **/move\_bb8\_in\_circle\_custom** service, BB-8 moves accordingly.
- Create a new python code called **call\_bb8\_move\_custom\_service\_server.py** that calls the service **/move\_bb8\_in\_circle\_custom**. Remember how it was done in **Unit 5 Services Part 1**.

Then, generate a new launch file called **call\_bb8\_move\_custom\_service\_server.launch** that executes the **call\_bb8\_move\_custom\_service\_server.py** through a node.

- End of Exercise 6.3 -

## 6.3 Summary

Let's do a quick summary of the most important parts of **ROS Services**, just to try to put everything in place.

A **ROS Service** provides a certain functionality of your robot. A ROS Service is composed of 2 parts:

- **Service Server**: This is what **PROVIDES** the functionality. Whatever you want your Service to do, you have to place it in the Service Server.
- **Service Client**: This is what **CALLS** the functionality provided by the Service Server. That is, it **CALLS** the Service Server.

ROS Services use a special service message, which is composed of 2 parts:

- **Request:** The request is the part of the message that is used to CALL the Service. Therefore, it is sent by the Service Client to the Service Server.
- **Response:** The response is the part of the message that is returned by the Service Server to the Service Client, once the Service has finished.

**ROS Services are synchronous.** This means that whenever you CALL a Service Server, you have to wait until the Service has finished (and returns a response) before you can do other stuff with your robot.

## 6.4 Services Quiz



For evaluation, this quiz will ask you to perform different tasks. For each task, very **specific instructions** will be provided: name of the package, names of the launch files and Python scripts, topic names to use, etc.

It is **VERY IMPORTANT** that you strictly follow these instructions, since they will allow our automated correction system to properly grade your quiz and assign a score to it. If the names you use are different from the ones specified in the exam instructions, your exercise will be marked as **FAILED**, even though it works correctly.

- Upgrade the Python file called **bb8\_move\_custom\_service\_server.py**. Modify the code you used in **Exercise 6.3**, which contained a Service Server that accepted a custom Service message to activate the circle movement (with a defined duration). This new service will be called **/move\_bb8\_in\_square\_custom**. This new service will have to use service messages of the **BB8CustomServiceMessage** type, which is defined here:

The **BB8CustomServiceMessage.srv** will be something like this:

```
In [ ]: float64 side      # The distance of each side of the square
        int32 repetitions # The number of times BB-8 has to execute the square movement when the service is called
        ---
        bool success      # Did it achieve it?
```

**NOTE:** The **side** variable doesn't represent the real distance along each size of the square. It's just a variable that will be used to change the size of the square. The bigger the **size** variable is, the bigger the square performed by the BB-8 robot will be.

- In the previous exercises, you were triggering a circle movement when calling to your service. In this new service, the movement triggered will have to be a square, like in the image below:

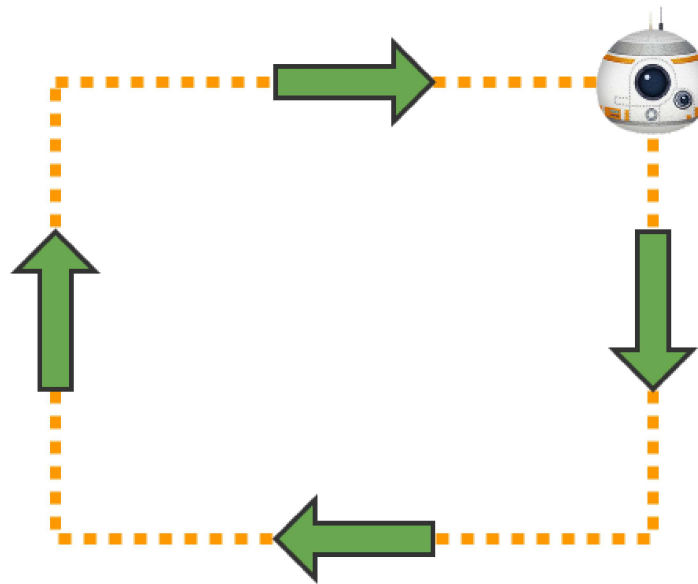


Fig.3.1 - BB8 Square Movement Diagram

**NOTE:** Keep in mind that due to the inertias of the robot it can be hard to perform a perfect square movement. But this is not the goal of this Exercise. The square movement **DOES NOT HAVE TO BE PERFECT**. It just has to be a square-like movement.

- Use the data passed to this new `/move_bb8_in_square_custom` to change the way BB-8 moves.

Depending on the **side** value, the service must move the BB-8 robot in a square movement based on the **side** given.

Also, the BB-8 must repeat the shape as many times as indicated in the **repetitions** variable of the message.

Finally, it must return **True** if everything went OK in the **success** variable.

**NOTE:** The **side** variable doesn't represent the real distance along each side of the square. It's just a variable that will be used to change the size of the square. The bigger the **side** variable is, the bigger the square performed by the BB-8 robot will be.

- Create a new launch file, called `start_bb8_move_custom_service_server.launch`, that launches the new `bb8_move_custom_service_server.py` file.
- Test that when calling this new `/move_bb8_in_square_custom` service, BB-8 moves accordingly. This means, the square is performed taking into account the **side** and **repetitions** variables.
- Create a new service client that calls the service `/move_bb8_in_square_custom`, and makes BB-8 move in a small square **twice** and in a bigger square **once**. It will be called `bb8_move_custom_service_client.py`. The launch that starts it will be called `call_bb8_move_in_square_custom_service_server.launch`.

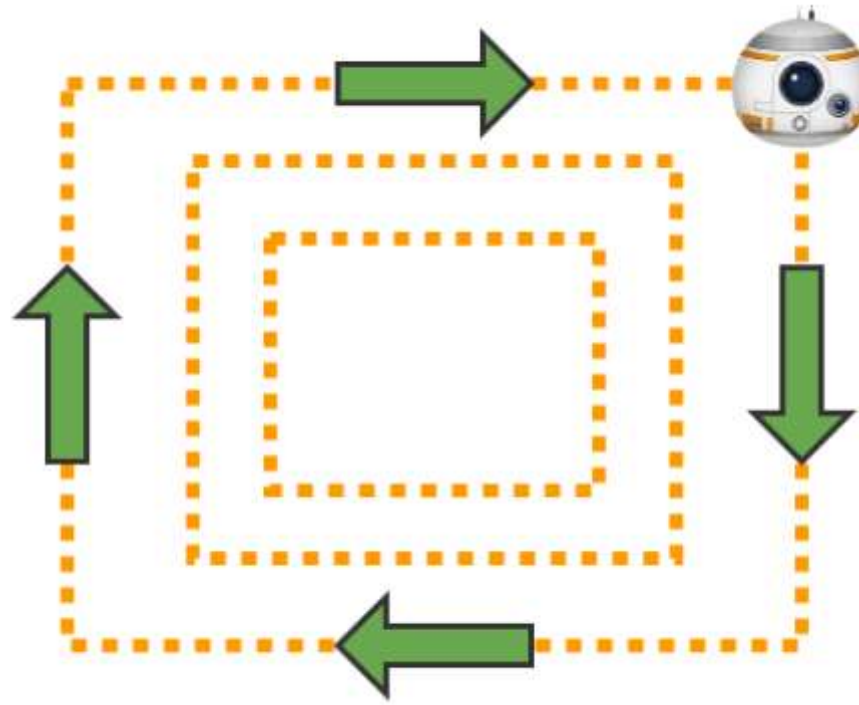
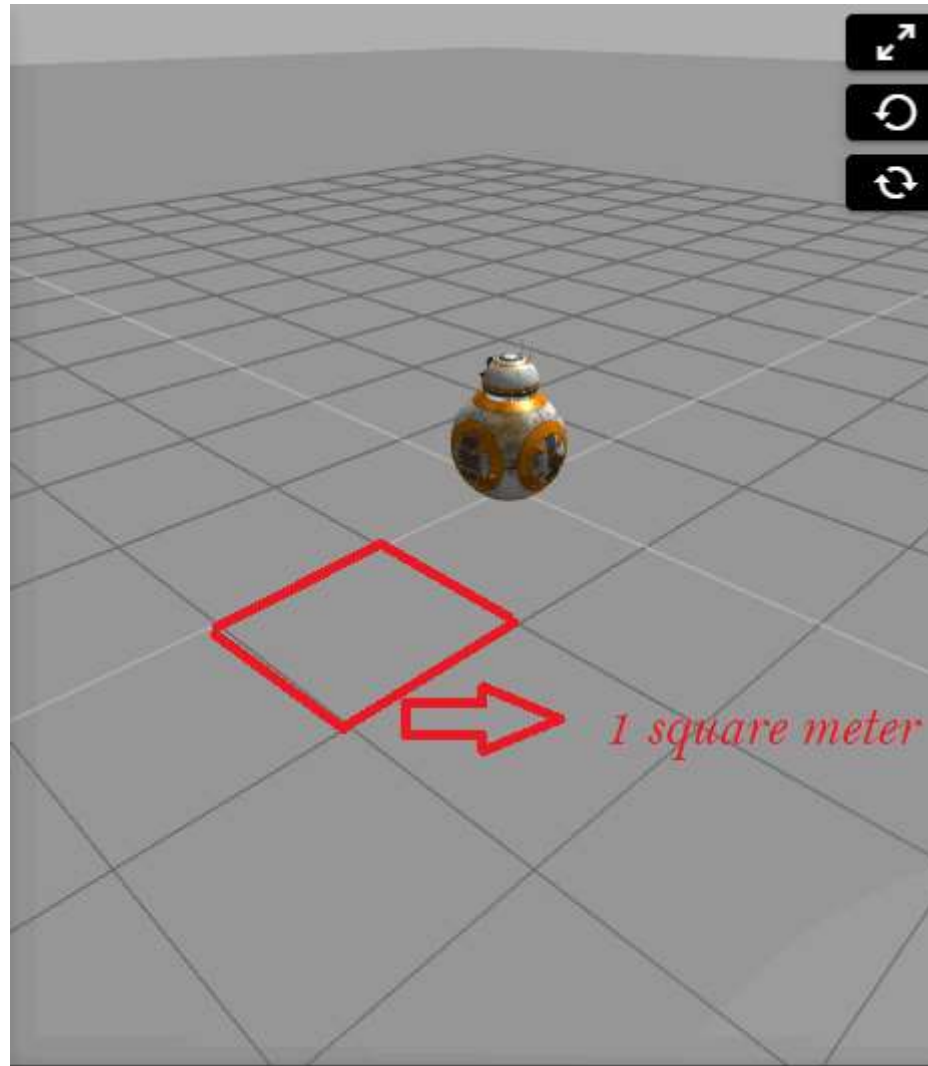


Fig.3.2 - BB8 Dynamic Square Diagram

- The **small square** has to be of, at least, **1 sqm**. The **big square** has to be of, at least, **2 sqm**.
- Please take a square on the floor "tiles" in the image below (as seen on the simulation) as **1 sqm**.



Square Meter Specification

Specifications

- The name of the package where you'll place all the code related to the quiz will be **services\_quiz**.
- The name of the launch file that will start your Service Server will be `start_bb8_move_custom_service_server.launch` . **This should only start the service server, and not the service client.**
- The name of the service will be `/move_bb8_in_square_custom`.
- The name of your Service message file will be **BB8CustomServiceMessage.srv**.
- The Service message file will be placed at the **services\_quiz** package, as indicated in the 1st point.
- The name of the launch file that will start your Service Client will be `call_bb8_move_in_square_custom_service_server.launch` . **This launch file should not start the Service Server, only the Service Client.**
- Your service client script, `bb8_move_custom_service_client.py` , **must exit cleanly** after it completes the movements. Please also **ensure that it completes the movements within THREE minutes**.
  - If it does not exit after three minutes, it will be killed automatically and you may not get credit for work done even if it works as expected.
  - Do not use `rospy.spin()` as it will block the script from exiting.
- The **small square** has to be of, at least, **1 sqm**. The **big square** has to be of, at least, **2 sqm**.
- Before correcting your Quiz, make sure that all your Python scripts are executable. They need to have full execution permissions in order to be executed by our autocorrection system. You can give them full execution permissions with the following command:

In [ ]: `chmod +x my_script.py`



- Before correcting your Quiz, make sure you have terminated all the programs in your Web Shells.

The following will be checked, in order. *If a step fails, the steps following are skipped.*

1. Does the package exist?
2. Did the package compile successfully?
3. Is the custom service message created successfully?
4. Is the service server started by the service server launch file?
5. Is the service client started by the service client launch file?
6. Did the robot perform the expected movements?

The grader will provide as much feedback on any failed step so you can make corrections where necessary.

#### Quiz Correction

When you have finished the quiz, you can correct it in order to get a score. For that, just click on the following button at the top of this Notebook.





Final Mark

In case you fail the Quiz, or you don't get the desired mark, do not get frustrated! You will have the chance to resend the Quiz in order to improve your score.



Congratulations! You are now ready to add all of the services that you want to your own personal astromech droid!

**You should finish PART II of the project before attempting next unit of this course!**

