



2021

**INFO 802**

**Master Advanced Mechatronics**

Luc Marechal



**ROS**

**ROS message, ROS Service  
Lecture 3**

# ROS Command Tools

See message definition information:

```
> rosmmsg show [message_type]
```

See active topics:

```
> rostopic list
```

See node information:

```
> rosnode info [message_type]
```

# ROS System File Commands

Get information on packages

```
> rospack find [package_name]
```


Change directory (cd) directly to a package or a stack

```
> roscd [location_name[/subdir]]
```

/s directly in a package by name rather than by absolute path

```
> rosls [location_name[/subdir]]
```

## ROS CHEAT SHEET MELODIC

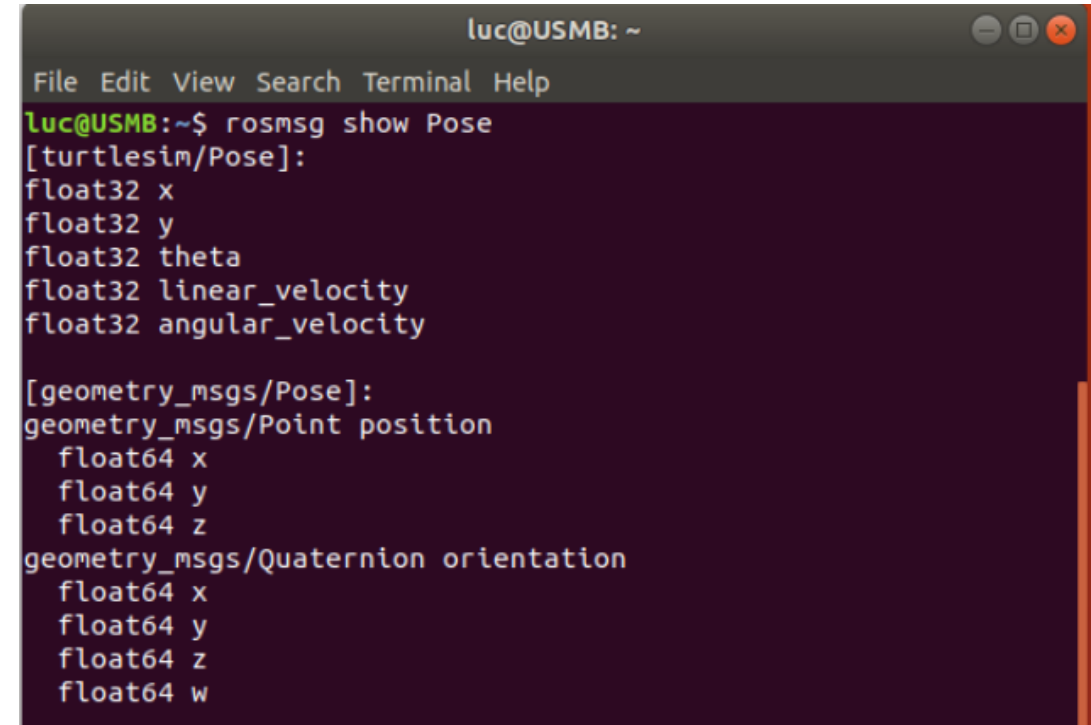
WORKSPACES	CMakeLists.txt	RUNNING SYSTEM
<b>Create Workspace</b> mkdir catkin_ws && cd catkin_ws wstool init src catkin_make source devel/setup.bash	<b>Skeleton</b> cmake_minimum_required(VERSION 2.8.3) project(package_name) find_package(catkin REQUIRED) catkin_package()	Run ROS using plain: roscore  Alternatively, roslaunch will run its own roscore automatically if it can't find one: roslaunch my_package package_launchfile.launch  Suppress this behaviour with the --wait flag.
<b>Add Repo to Workspace</b> roscd; cd ../src wstool set repo_name \ --git http://github.com/org/repo_name.git \ --version-melodic-devel wstool up	<b>Package Dependencies</b> To use headers or libraries in a package, or to use a package's exported CMake macros, express a build-time dependency: find_package(catkin REQUIRED COMPONENTS roscpp)  Tell dependent packages what headers or libraries to pull in when your package is declared as a catkin component: catkin_package( INCLUDE_DIRS include LIBRARIES \${PROJECT_NAME} CATKIN_DEPENDS roscpp)	<b>Nodes, Topics, Messages</b> roscd roscd list rostopic list rostopic echo cmd_vel rostopic hz cmd_vel rostopic info cmd_vel rostopic show geometry_msgs/Twist
<b>Resolve Dependencies in Workspace</b> sudo rosdep init # only once rosdep update rosdep install --from-paths src --ignore-src \ --rosdistro=\${ROS_DISTRO} -y	<b>PACKAGES</b> <b>Create a Package</b> catkin_create_pkg package_name [dependencies ...]  <b>Package Folders</b> include/package_name C++ header files  src Source files Python libraries in subdirectories  scripts Python nodes and scripts  msg, srv, action Message, Service, and Action definitions	<b>Remote Connection</b> Master's ROS environment: <ul style="list-style-type: none"><li>ROS_IP or ROS_HOSTNAME set to this machine's network address.</li><li>ROS_MASTER_URI set to URI containing that IP or hostname.</li></ul> Your environment: <ul style="list-style-type: none"><li>ROS_IP or ROS_HOSTNAME set to your machine's network address.</li><li>ROS_MASTER_URI set to the URI from the master.</li></ul> To debug, check ping from each side to the other, run roswtf on each side.
<b>Release Repo Packages</b> catkin_generate_changelog # review & commit changelogs catkin_prepare_release bloom-release --track melodic --ros-distro melodic repo_name	<b>Messages, Services</b> These go after find_package(), but before catkin_package(). Example: find_package(catkin REQUIRED COMPONENTS message_generation std_msgs) add_message_files(FILES MyMessage.msg) add_service_files(FILES MyService.msg) generate_messages(DEPENDENCIES std_msgs) catkin_package(CATKIN_DEPENDS message_runtime std_msgs)w	<b>ROS Console</b> Adjust using rqt_logger_level and monitor via rqt_console. To enable debug output across sessions, edit the \$HOME/.ros/config/rosconsole.config and add a line for your package: log4j.logger.\${ros.package_name}=DEBUG  And then add the following to your session: export ROSCONSOLE_CONFIG_FILE=\$HOME/.ros/config/rosconsole.config  Use the roslaunch --screen flag to force all node output to the screen, as if each declared <node> had the output="screen" attribute.
<b>Reminders</b> <ul style="list-style-type: none"><li>Testable logic</li><li>Publish diagnostics</li><li>Desktop dependencies in a separate package</li></ul>	<b>Build Libraries, Executables</b> Goes after the catkin_package() call. add_library(\${PROJECT_NAME} src/main) add_executable(\${PROJECT_NAME}_node src/main) target_link_libraries( \${PROJECT_NAME}_node \${catkin_LIBRARIES})  <b>Installation</b> install(TARGETS \${PROJECT_NAME} DESTINATION \${CATKIN_PACKAGE_LIB_DESTINATION}) install(TARGETS \${PROJECT_NAME}_node DESTINATION \${CATKIN_PACKAGE_BIN_DESTINATION}) install(PROGRAMS scripts/myScript DESTINATION \${CATKIN_PACKAGE_BIN_DESTINATION}) install(DIRECTORY launch DESTINATION \${CATKIN_PACKAGE_SHARE_DESTINATION})	 www.clearpathrobotics.com/ros-cheat-sheet © 2019 Clearpath Robotics, Inc. All Rights Reserved.

More info  
<http://wiki.ros.org/ROS/Tutorials/NavigatingTheFilesystem>

# ROS Command Tools

See message definition information:

```
> rosmmsg show [message_type]
```



```
luc@USMB: ~  
File Edit View Search Terminal Help  
luc@USMB:~$ rosmmsg show Pose  
[turtlesim/Pose]:  
float32 x  
float32 y  
float32 theta  
float32 linear_velocity  
float32 angular_velocity  
  
[geometry_msgs/Pose]:  
geometry_msgs/Point position  
  float64 x  
  float64 y  
  float64 z  
geometry_msgs/Quaternion orientation  
  float64 x  
  float64 y  
  float64 z  
  float64 w
```

The message of type *Pose* is defined in the package *turtlesim* but also in the package *geometry\_msgs* but they are not the same !

# ROS computation graph *rqt*

Visualize running topics and nodes

```
> rosrun rqt_graph rqt_graph
```

Visualize running topics and nodes

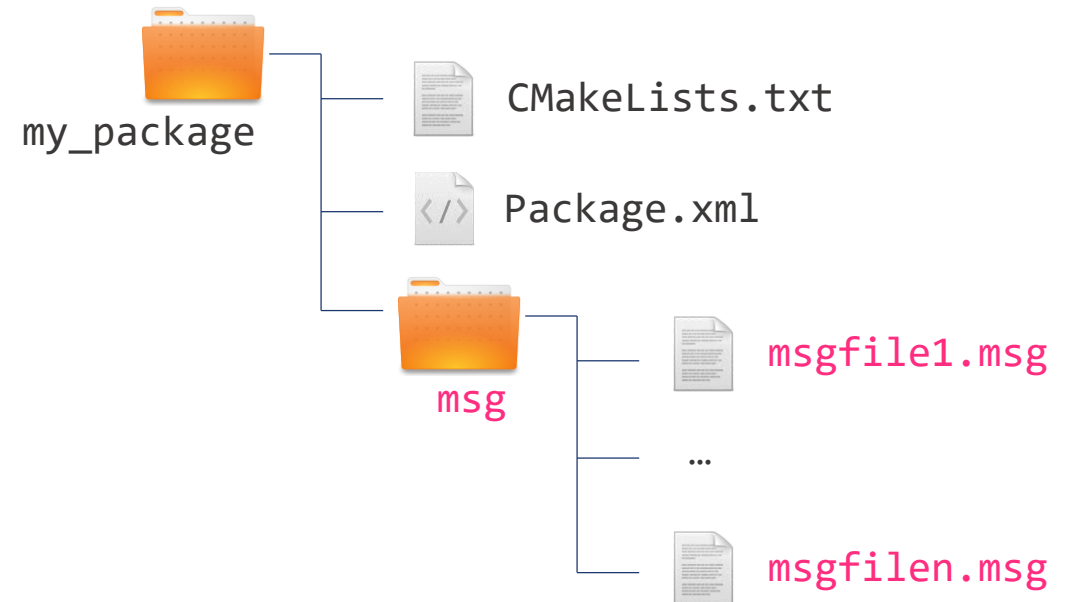
```
> rosrun rqt_plot rqt_plot
```

# ROS Messages

- They are files where we put a specification about the type of data to be transmitted and the values of this data.
- Defined in *\*.msg* files stored in the msg subdirectory of a package

See message definition information with

```
> rosmmsg show [message_type]
```



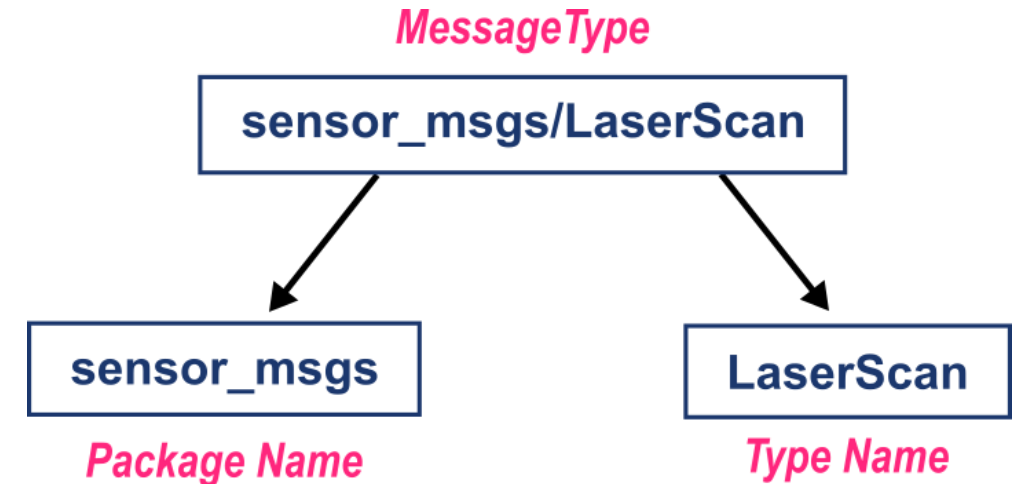
# ROS Messages

- Every message type belongs to a specific package

Message type names always contain a slash, and the part before the slash is the name of the containing package:

```
> package_name/type_name
```

*Example:*



# ROS Messages

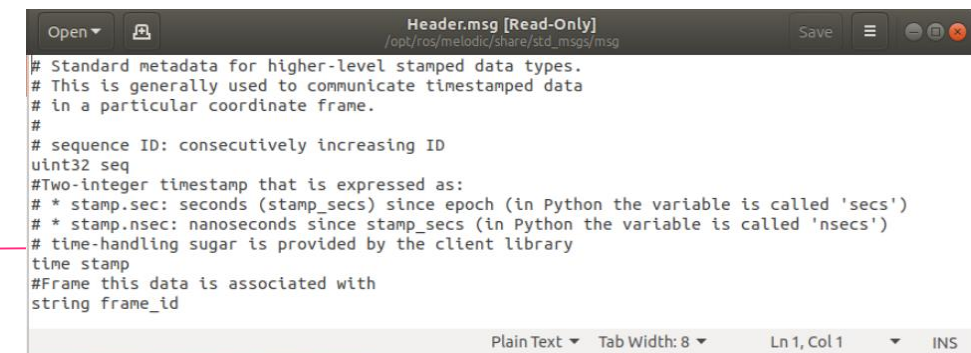
- msgs are just simple text files with a field type and field name per line. The field types you can use are:

- int8, int16, int32, int64 (plus uint\*)
- float32, float64
- string
- time, duration
- other msg files
- variable-length array[] and fixed-length array[C]

- Header: special type in ROS

The header contains a timestamp and coordinate frame information that are commonly used in ROS to communicate timestamped data in a particular coordinate frame.

```
uint32 seq  
time stamp  
string frame_id
```



The screenshot shows a text editor window titled "Header.msg [Read-Only]" with the file path "/opt/ros/melodic/share/std\_msgs/msg". The content of the file is as follows:

```
# Standard metadata for higher-level stamped data types.  
# This is generally used to communicate timestamped data  
# in a particular coordinate frame.  
#  
# sequence ID: consecutively increasing ID  
uint32 seq  
#Two-integer timestamp that is expressed as:  
# * stamp.sec: seconds (stamp_secs) since epoch (in Python the variable is called 'secs')  
# * stamp.nsec: nanoseconds since stamp_secs (in Python the variable is called 'nsecs')  
# time-handling sugar is provided by the client library  
time stamp  
#Frame this data is associated with  
string frame_id
```

The editor interface includes a menu bar with "Open", "Save", and other icons. The status bar at the bottom indicates "Plain Text", "Tab Width: 8", "Ln 1, Col 1", and "INS".



# ROS Messages

- Standard type to use in message

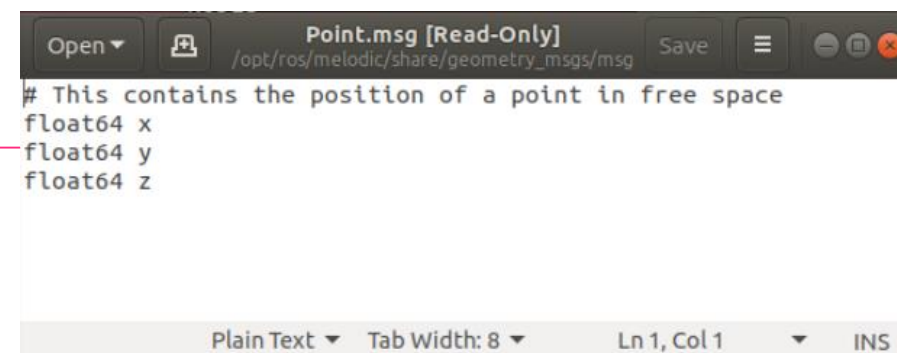
Primitive type	Serialization	C++	Python
bool	Unsigned 8-bit int	uint8_t	bool
int8	Signed 8-bit int	int8_t	int
uint8	Unsigned 8-bit int	uint8_t	int
int16	Signed 16-bit int	int16_t	int
uint16	Unsigned 16-bit int	uint16_t	int
int32	Signed 32-bit int	int32_t	int
uint32	Unsigned 32-bit int	uint32_t	int
int64	Signed 64-bit int	int64_t	long
uint64	Unsigned 64-bit int	uint64_t	long
float32	32-bit IEEE float	float	float
float64	64-bit IEEE float	double	float
string	ASCII string (4-bit)	std::string	string
time	Secs/nsecs signed 32-bit ints	ros::Time	rospy. Time
duration	Secs/nsecs signed 32-bit ints	ros::Duration	rospy. Duration

# ROS Messages

- Examples

geometry\_msgs/Points.msg

```
float64 x  
float64 y  
float64 z
```



The screenshot shows a text editor window titled "Point.msg [Read-Only]" with the file path "/opt/ros/melodic/share/geometry\_msgs/msg". The editor contains the following text: "# This contains the position of a point in free space", "float64 x", "float64 y", and "float64 z". The status bar at the bottom indicates "Plain Text", "Tab Width: 8", "Ln 1, Col 1", and "INS".

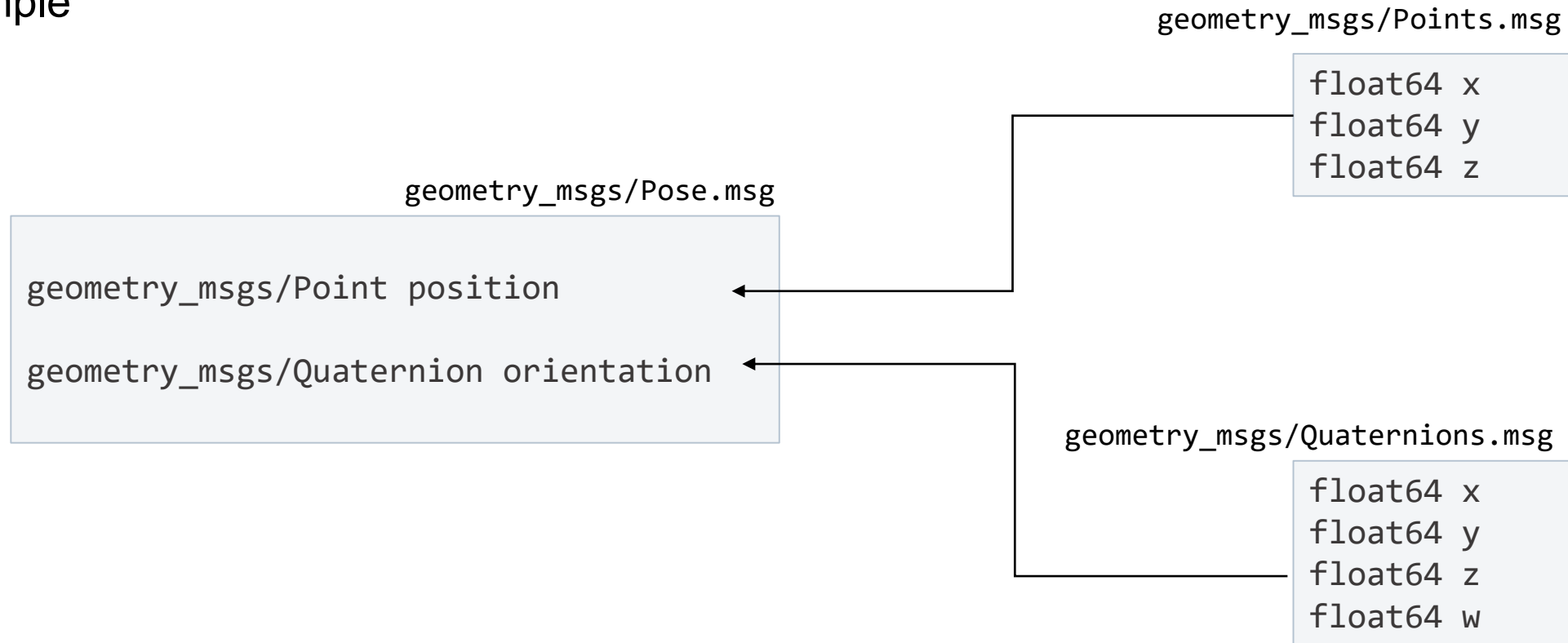
```
# This contains the position of a point in free space  
float64 x  
float64 y  
float64 z
```

geometry\_msgs/Quaternions.msg

```
float64 x  
float64 y  
float64 z  
float64 w
```

# ROS Messages

- Example



You can use message type from already existing message

# How to use ROS Messages in code?

Import the message type from the msg library

Use the message directly with an oriented object way

Use the message with an Object

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

```
from geometry_msgs.msg import Pose  
from beginner_tutorials.msg import My_Custom_Message
```

```
# without creating an object  
Pose.position.x = 1.0
```

```
# by creating an object
```

```
My_Object = Pose()  
My_Object.position.x = 1.0
```

```
My_Object.orientation.y = My_Object.position.x + 43.2
```

```
[geometry_msgs/Pose]:  
geometry_msgs/Point position  
float64 x  
float64 y  
float64 z  
geometry_msgs/Quaternion orientation  
float64 x  
float64 y  
float64 z  
float64 w
```

# Creating a custom ROS msg

## Edit \*.msg file

- When Should You Make a New Message Type?

Only when you absolutely have to (check before with *rosmmsg* to see if there is already something there that you can use instead).

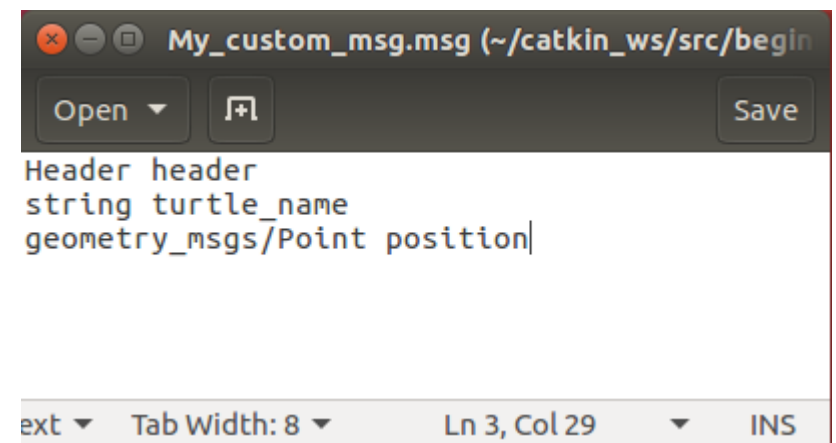
However, there are times when the built-in message types are not enough, and we have to define our own messages

Create a subfolder named **msg** in your package folder

```
> cd ~/catkin_ws/src/beginner_tutorials  
> mkdir msg
```

Create a new **my\_custom\_msg.msg** file and add the following lines

```
> subl msg/my_custom_msg.msg
```



# Creating a custom ROS msg

## Modify package.xml file

- We need to make sure that the msg files are turned into source code for C++, Python, and other languages

uncomment those two lines in the package.xml file

```
<build_depend>message_generation</build_depend>  
<run_depend>message_runtime</run_depend>
```

- Note that at build time, we need "message\_generation", while at runtime, we need "message\_runtime"

# Creating a custom ROS msg

## Modify CMakefile.txt file

- In CMakeLists.txt add the message\_generation dependency to the find package call so that you can generate messages:

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

- Also make sure you export the message runtime dependency:

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

# Creating a custom ROS msg

## Modify CMakefile.txt file

- Find the following block

```
## Generate messages in the 'msg' folder
# add_message_files(
#   FILES
#   Message1.msg
#   Message2.msg
# )
```

- Uncomment it by removing the # symbols and then replace the stand in Message\*.msg files with your .msg file, such that it looks like this:

```
add_message_files(
  FILES
  my_custom_msg.msg
)
```



# Creating a custom ROS msg

## Modify CMakefile.txt file

- ensure the generate\_messages() function is called: uncomment this lines

```
# generate_messages(  
#   DEPENDENCIES  
#   std_msgs  
# )
```

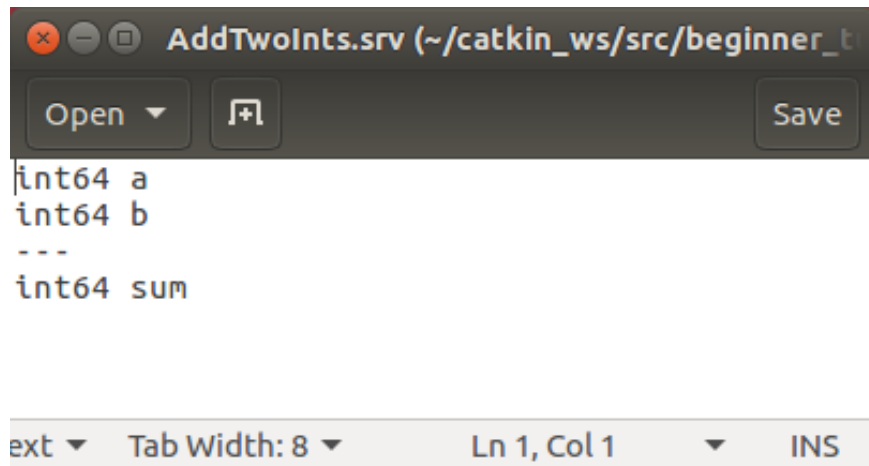
- So it looks like:

```
generate_messages(  
  DEPENDENCIES  
  std_msgs  
)
```

cmake will then know that the project needs to be reconfigured with the addition of msg files

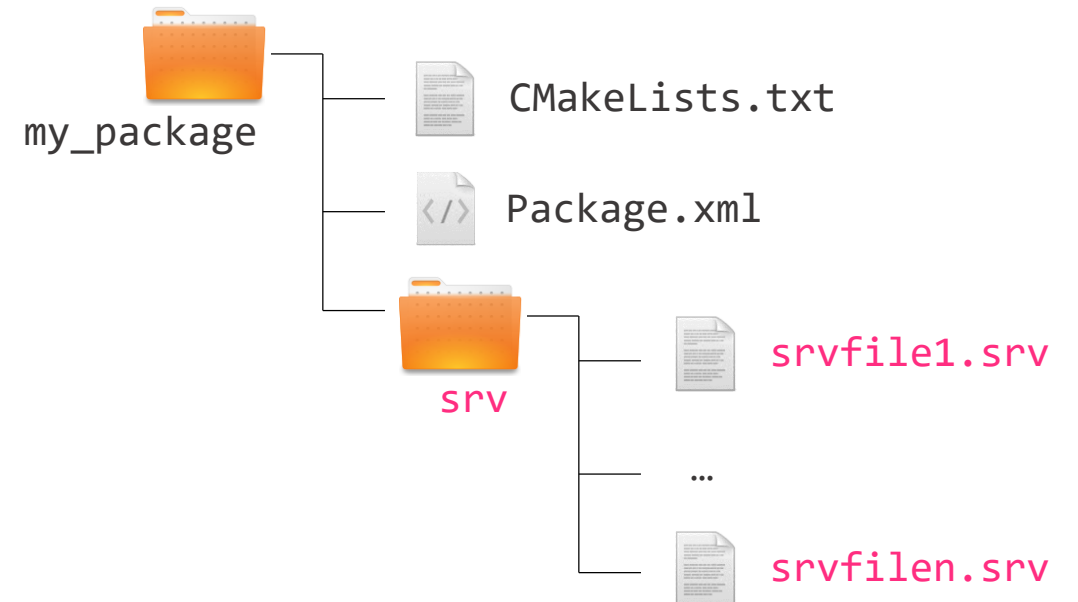
## Creating a ROS srv

- Defined in `*.srv` files stored in the `srv` subdirectory of a package
- `srv` files are just like `msg` files, except they contain two parts: a request and a response. The two parts are separated by a `'---'` line.



```
int64 a
int64 b
---
int64 sum
```

ext Tab Width: 8 Ln 1, Col 1 INS



# Creating a ROS srv

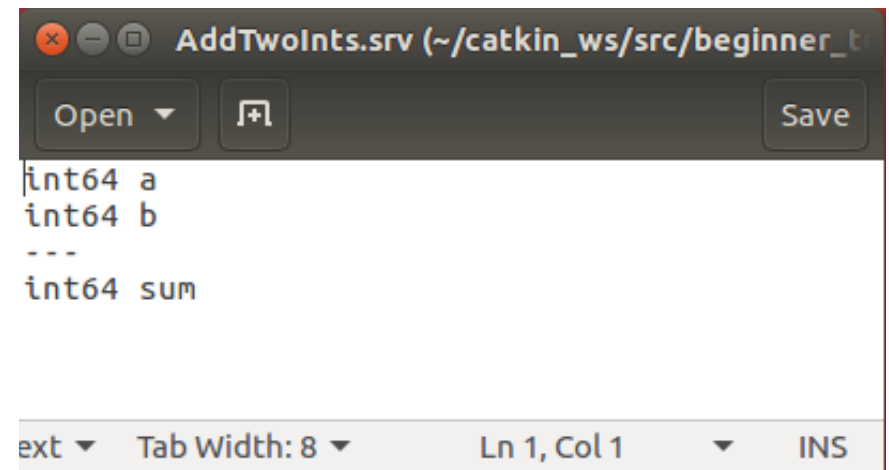
## Modify package.xml file

Create a subfolder named **srv** in your package folder

```
> cd ~/catkin_ws/src/beginner_tutorials  
> mkdir srv
```

Example: create `AddTwoInts.srv` file and add the following lines

```
> subl srv/AddTwoInts.srv
```



The screenshot shows a text editor window titled "AddTwoInts.srv (~/.catkin\_ws/src/beginner\_tutorials)". The window has a dark theme and includes "Open", "Save", and a "New" icon button. The content of the file is as follows:

```
int64 a  
int64 b  
---  
int64 sum
```

At the bottom of the window, the status bar displays "ext", "Tab Width: 8", "Ln 1, Col 1", and "INS".

# Creating a ROS srv

## Modify package.xml file

- We need to make sure that the srv files are turned into source code for C++, Python, and other languages

uncomment those two lines in the package.xml file

```
<build_depend>message_generation</build_depend>  
<run_depend>message_runtime</run_depend>
```

- Note that at build time, we need "message\_generation", while at runtime, we need "message\_runtime"

# Creating a ROS srv

## Modify CMakefile.txt file

- In CMakeLists.txt add the message\_generation dependency to the find package call so that you can generate messages:

(Despite its name, message\_generation works for both msg and srv.)

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

# Creating a ROS srv

## Modify CMakefile.txt file

- Find the following block

```
## Generate messages in the 'msg' folder
# add_service_files(
#   FILES
#   Service1.msg
#   Service2.msg
# )
```

- Uncomment it by removing the # symbols and then replace the stand in Service\*.msg files with your .srv file, such that it looks like this:

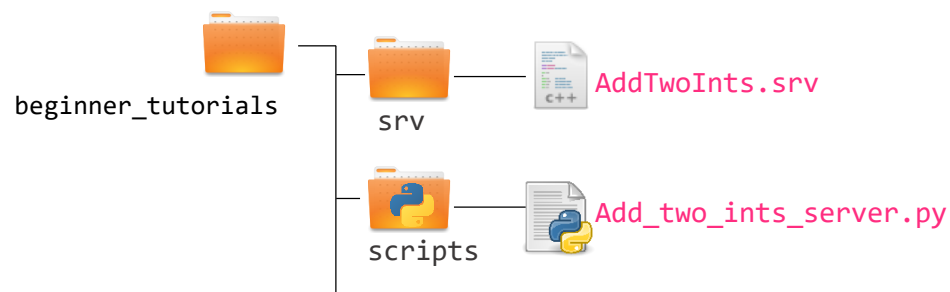
```
add_service_files(
  FILES
  AddTwoInts.srv
)
```

# Creating a Service and Client Node (Python)

## Writing the **Service** Node

Edit a py file in scripts folder

```
> cd ~/catkin_ws/beginner_tutorials/scripts  
> sudo subl Add_two_ints_server.py
```



... and make it executable

```
> sudo chmod +x Add_two_ints_server.py
```

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

from beginner_tutorials.srv import *
import rospy

def handle_add_two_ints(req):
    print "Returning [%s + %s = %s]"%(req.a, req.b, (req.a + req.b))
    return AddTwoIntsResponse(req.a + req.b)

def add_two_ints_server():
    rospy.init_node('add_two_ints_server')
    s = rospy.Service('add_two_ints', AddTwoInts,
        handle_add_two_ints)
    print "Ready to add two ints."
    rospy.spin()

if __name__ == "__main__":
    add_two_ints_server()
```

# Creating a Service and Client Node (Python)

## Examining the **Service** Node

The *service* file has been defined and is located in the *srv* folder

*init\_node()*: declare the node

This declares a new service named *add\_two\_ints* with the *AddTwoInts* service type. All requests are passed to *handle\_add\_two\_ints* function. *handle\_add\_two\_ints* is called with instances of *AddTwoIntsRequest* and returns instances of *AddTwoIntsResponse*.

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

from beginner_tutorials.srv import *
import rospy

def handle_add_two_ints(req):
    print "Returning [%s + %s = %s]"%(req.a, req.b, (req.a + req.b))
    return AddTwoIntsResponse(req.a + req.b)

def add_two_ints_server():
    rospy.init_node('add_two_ints_server')
    s = rospy.Service('add_two_ints', AddTwoInts, handle_add_two_ints)
    print "Ready to add two ints."
    rospy.spin()

if __name__ == "__main__":
    add_two_ints_server()
```

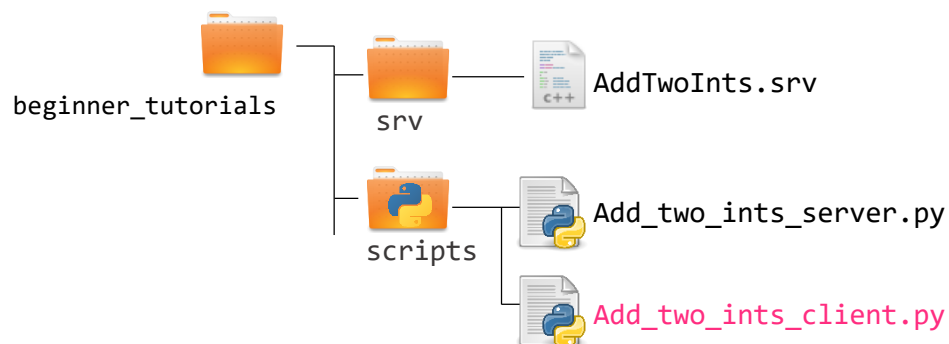


# Creating a Service and Client Node (Python)

## Writing the Client Node

Edit a py file in scripts folder

```
> cd ~/catkin_ws/beginner_tutorials/scripts  
> sudo subl Add_two_ints_client.py
```



... and make it executable

```
> sudo chmod +x Add_two_ints_client.py
```

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

import sys
import rospy
from beginner_tutorials.srv import *

def add_two_ints_client(x, y):
    rospy.wait_for_service('add_two_ints')
    try:
        add_two_ints = rospy.ServiceProxy('add_two_ints', AddTwoInts)
        resp1 = add_two_ints(x, y)
        return resp1.sum
    except rospy.ServiceException, e:
        print "Service call failed: %s"%e

def usage():
    return "%s [x y]"%sys.argv[0]

if __name__ == "__main__":
    if len(sys.argv) == 3:
        x = int(sys.argv[1])
        y = int(sys.argv[2])
    else:
        print usage()
        sys.exit(1)
    print "Requesting %s+%s"%(x, y)
    print "%s + %s = %s"%(x, y, add_two_ints_client(x, y))
```

# Creating a Service and Client Node (Python)

## Examining the Client Node

Wait for the service named `add_two_ints` to be advertised by the server

Once the service is advertised, we can set up a local proxy for it

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

import sys
import rospy
from beginner_tutorials.srv import *

def add_two_ints_client(x, y):
    rospy.wait_for_service('add_two_ints')
    try:
        add_two_ints = rospy.ServiceProxy('add_two_ints', AddTwoInts)
        resp1 = add_two_ints(x, y)
        return resp1.sum
    except rospy.ServiceException, e:
        print "Service call failed: %s"%e

def usage():
    return "%s [x y]"%sys.argv[0]

if __name__ == "__main__":
    if len(sys.argv) == 3:
        x = int(sys.argv[1])
        y = int(sys.argv[2])
    else:
        print usage()
        sys.exit(1)
    print "Requesting %s+%s"%(x, y)
    print "%s + %s = %s"%(x, y, add_two_ints_client(x, y))
```

# Creating a Publisher and a Subscriber Node (Python)

## Building the nodes

If not done yet: make the node executable (for Python only)

```
> subl chmod +x scripts/add_two_ints_server.py  
> subl chmod +x scripts/add_two_ints_client.py
```

### Build package

(we use Cmake as the build system even for Python nodes)

```
> cd ~/catkin_ws  
> catkin_make beginner_tutorials
```

Make sure you have sourced your workspace's setup.bash file

```
> source ~/catkin_ws/devel/setup.bash
```

# ROS Bags

- **rosvbag**: set of tools for recording messages and playing back later to ROS topics offline.
- Can be used to mimic real sensor streams for offline debugging.
- Useful for debugging algorithm.
- The file a name is in the format: *file\_name\_YYYY-MM-DD-HH-mm-ss.bag*

## Record topics with

```
> rosvbag record [topic_1] [topic_2] -o [bag_name]
```

## Playback messages with

```
> rosvbag play [bag_name]
```

## Examples

```
> rosvbag record -a
```

```
> rosvbag play --clock mybag.bag
```

```
luc@luc: ~/catkin_ws/bagfiles
luc@luc:~/catkin_ws/bagfiles$ rosvbag info 2019-03-07-18-03-41.bag
path:      2019-03-07-18-03-41.bag
version:   2.0
duration:  32.5s
start:     Mar 07 2019 18:03:41.15 (1551978221.15)
end:       Mar 07 2019 18:04:13.63 (1551978253.63)
size:      294.7 KB
messages:  4111
compression: none [1/1 chunks]
types:     geometry_msgs/Twist [9f195f881246fdfa2798d1d3eebca84a]
           rosvgraph_msgs/Log  [acffd30cd6b6de30f120938c17c593fb]
           turtlesim/Color      [353891e354491c51aabe32df673fb446]
           turtlesim/Pose       [863b248d5016ca62ea2e895ae5265cf9]
topics:    /rosout              4 msgs      : rosvgraph_msgs/Log
           /turtle1/cmd_vel      70 msgs      : geometry_msgs/Twist
           /turtle1/color_sensor 2022 msgs     : turtlesim/Color
           /turtle1/pose        2015 msgs     : turtlesim/Pose
```

# Example logging *turtlesim*

```
roscore http://USMB:11311/
```

```
PARAMETERS
```

```
* /rostdistro: kinetic  
* /rosversion: 1.12.14
```

```
NODES
```

```
auto-starting new master  
process[master]: started with pid [11443]  
ROS_MASTER_URI=http://USMB:11311/  
  
setting /run_id to cfd74b78-41a4-11e9-b61e-08002711f8f  
process[rosout-1]: started with pid [11456]  
started core service [/rosout]  
□
```

```
luc@USMB: ~  
luc@USMB:~$ rosrunc turtlesim turtle_teleop_key  
Reading from keyboard  
-----  
Use arrow keys to move the turtle.  
□
```

```
luc@USMB: ~  
luc@USMB:~$ □
```

```
luc@USMB: ~
```

```
luc@USMB:~$ rosrunc turtlesim turtlesim_node  
[rospack] Error: package 'turtlesim' not found  
luc@USMB:~$ rosrunc turtlesim turtlesim_node  
[ INFO] [1552051226.777712214]: Starting turtlesim with node name  
/turtlesim  
[ INFO] [1552051226.789758992]: Spawning turtle [turtle1] at x=[5  
.544445], y=[5.544445], theta=[0.000000]  
□
```

TurtleSim



## ROS computation graph *rqt*

- *rqt\_graph* creates a dynamic graph of what's going on in the system
- *rqt\_console* attaches to ROS's logging framework to display output from nodes. *rqt\_logger\_level* allows us to change the verbosity level (DEBUG, WARN, INFO, and ERROR) of nodes as they run.
- Prerequisite: Install *rqt* package

```
> sudo apt-get install ros-melodic-rqt ros-melodic-rqt-common-plugins
```

Launch *rqt\_console*

```
> rosrun rqt_console rqt_console
```

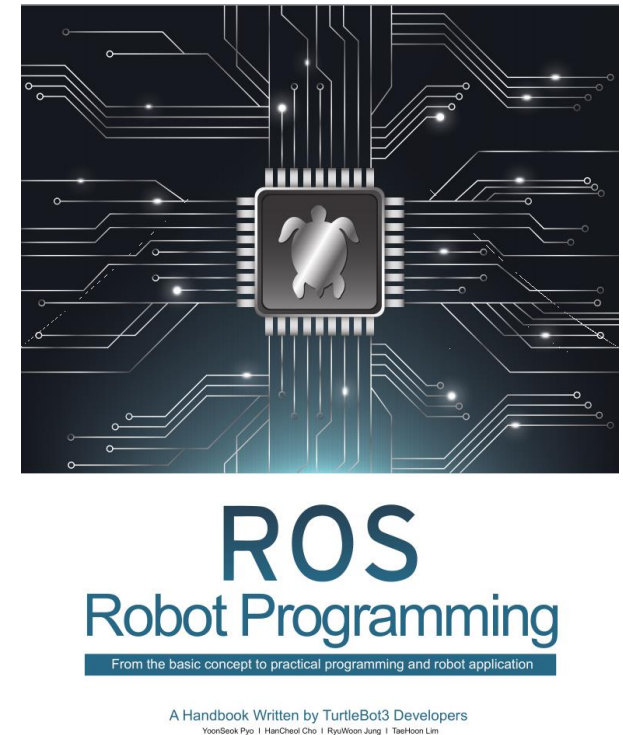
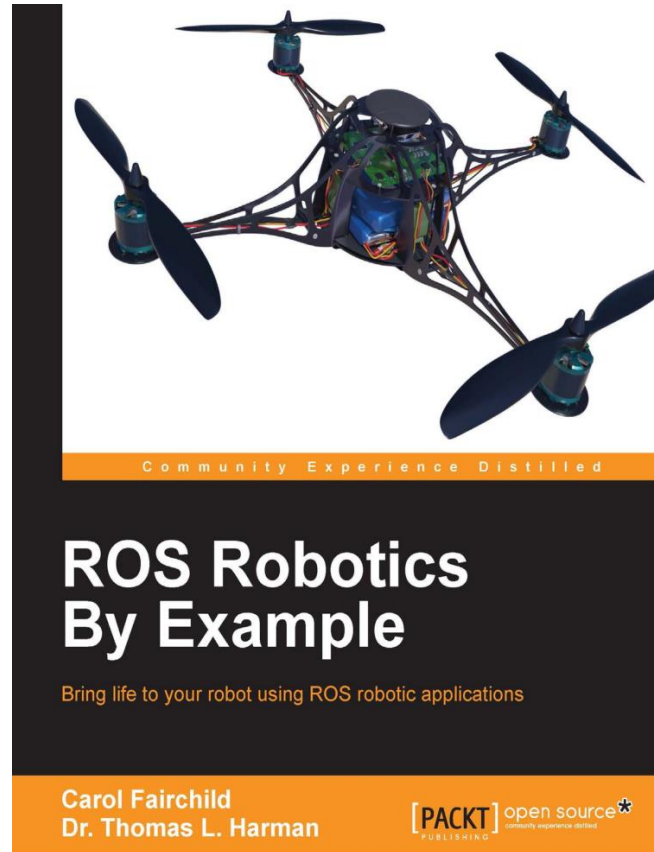
Launch *roslaunch rqt\_logger\_level rqt\_logger\_level* (in an other terminal)

```
> roslaunch rqt_logger_level rqt_logger_level
```

## Further References

- **ROS Wiki**
  - <http://wiki.ros.org/>
- **Installation**
  - <http://wiki.ros.org/ROS/Installation>
- **Tutorials**
  - <http://wiki.ros.org/ROS/Tutorials>
- **Available packages**
  - <http://www.ros.org/browse/>
- **ROS Cheat Sheet**
  - <https://www.clearpathrobotics.com/ros-robot-operating-system-cheat-sheet/>
  - [https://kapeli.com/cheat\\_sheets/ROS.docset/](https://kapeli.com/cheat_sheets/ROS.docset/)
- **ROS Best Practices**
  - [https://github.com/leggedrobotics/ros\\_best\\_practices/wiki](https://github.com/leggedrobotics/ros_best_practices/wiki)
- **ROS Package Template**
  - [https://github.com/leggedrobotics/ros\\_best\\_practices/tree/master/ros\\_package\\_template](https://github.com/leggedrobotics/ros_best_practices/tree/master/ros_package_template)

## Relevant books





## Contact Information

### Université Savoie Mont Blanc

Polytech' Annecy Chambéry  
Chemin de Bellevue  
74940 Annecy  
France

<https://www.polytech.univ-savoie.fr>

### Lecturer

Luc Marechal (luc.marechal@univ-smb.fr)  
SYMME Lab (Systems and Materials for Mechatronics)



SYMME