

ROS COMMANDS

ROBOTICS



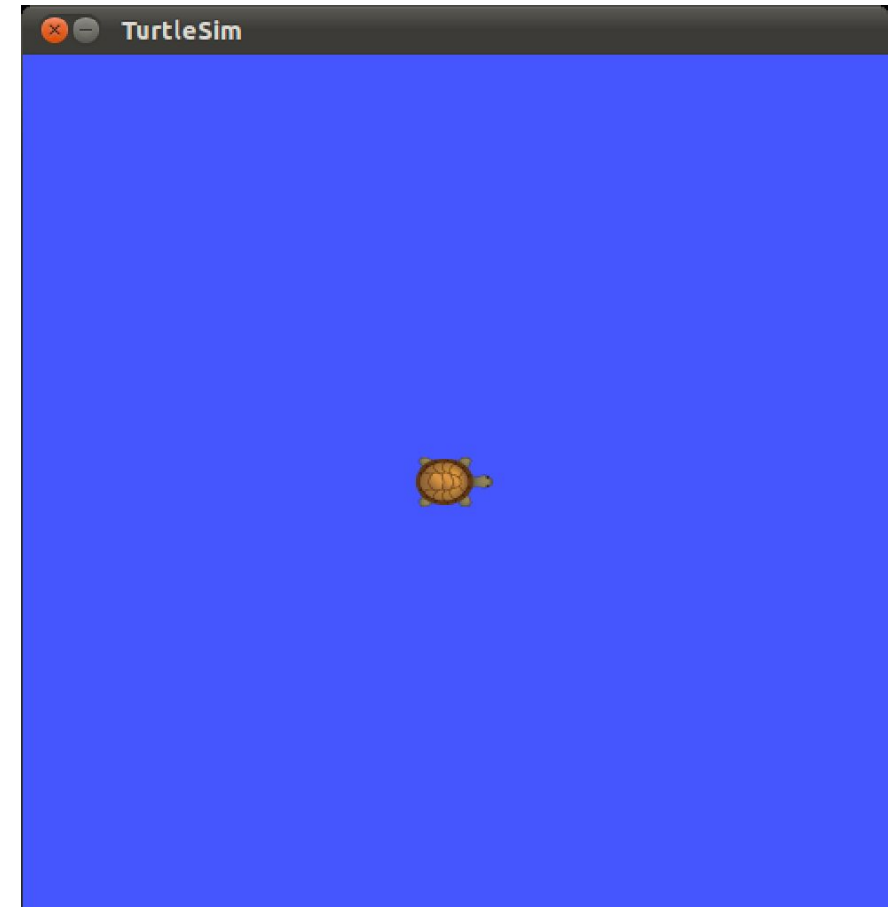
POLITECNICO
MILANO 1863

FILE SYSTEM TOOLS



Ros Desktop-full come with lots of tutorials and tools

Before creating our own package and start writing some code we will learn how to navigate the ROS file system and use the turtlesim package to test some of the most useful tools





FILE SYSTEM TOOLS

Change directory in the ROS file system

roscd [package_name[/subdir]]

roscd roscpp && pwd /opt/ros/kinetic/share/roscpp

roscd roscpp/srv /opt/ros/kinetic/share/roscpp/srv

roscd robby_roboto ~/catkin_ws/src/robby_roboto

FILE SYSTEM TOOLS



Getting information about installed packages

rospack <subcommand> [options] [package]

subcommands (among the others)

depends [package] package dependencies

find [package] find package directory

list list available packages

profile scan all workspace and index packages

rospack find roscpp /opt/ros/kinetic/share/roscpp

rospack list <several packages>

STARTING THE MIDDLEWARE



To start the ROS middleware just type in a terminal

```
roscore
```

Now it is possible to display information about the elements currently running

```
roscnode list
```

```
rostopic list
```

```
rostopic echo /rosout
```

```
rosservice list
```

```
rqt_graph
```



DEALING WITH NODES

Getting information about running nodes

roscall <command> [other_commands]

subcommands (among the others)

ping test connectivity to node

info print information about node

kill kill a running node

cleanup purge registration information of unreachable nodes

roscall list

roscall info /rosout

STARTING ROS NODES

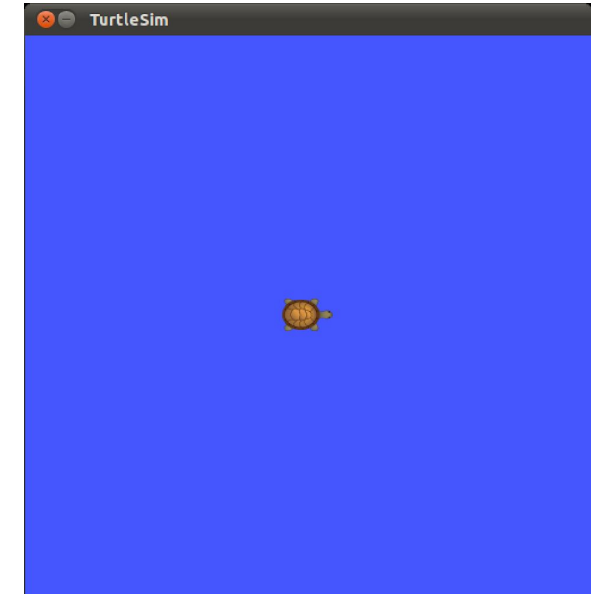


To start a ROS node type in a terminal
roslaunch [package_name] [node_name]

```
roslaunch turtlesim turtlesim_node
```

```
rostopic pub /turtle1/position 1.0 1.0 1.0
```

```
rostopic info /turtle1/position
```



/turtlesim

STARTING ROS NODES



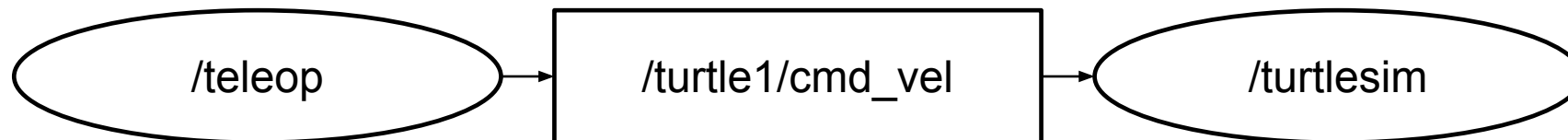
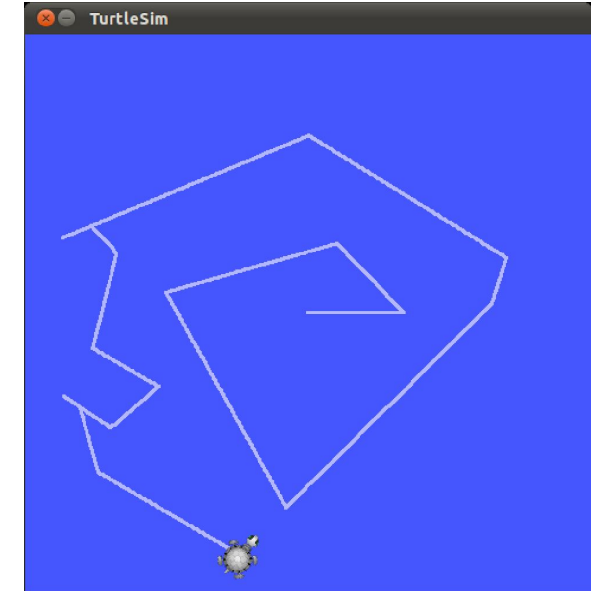
In a new terminal

```
roslaunch turtlesim turtle_teleop_key
```

Notes:

`turtle_teleop_key` is publishing the key strokes on a topic

`turtlesim` subscribes to the same topic to receive the key strokes



DEALING WITH TOPICS



To show the running node type in a terminal

```
rqt_graph
```

To plot published data on a topic

```
rqt_plot /turtle1/pose/x /turtle1/pose/y
```

```
rqt_plot /turtle1/pose/x:y
```

To monitor a topic on a terminal type

```
rostopic echo /turtle1/cmd_vel
```

DEALING WITH TOPICS CONT.



Getting information about ROS topics

rostopic <command> [topic_name]

subcommands (among the others)

echo print messages to screen

find find topics by type

hz display publishing rate of topic

info print information about active topic

list list active topics

pub publish data to topic

type print topic type



DEALING WITH TOPICS CONT.

Getting information about ROS topics

```
rostopic type [topic_name]
```

```
rostopic type /turtle1/cmd_vel
```

Publishing ROS topics

```
rostopic pub [topic] [msg type] [args]
```

```
$ rostopic pub my_topic std_msgs/String "hello there"
```

```
$ rostopic pub -r 10 /cmd_vel geometry_msgs/Twist '{linear: {x: 2.0, y: 0.0, z: 0.0},  
angular: {x: 0.0,y: 0.0,z: 0.0}}'
```



DEALING WITH TOPICS CONT.

```
$ rostopic pub -1 /turtle1/cmd_vel geometry_msgs/Twist -- '[2.0, 0.0, 0.0]' '[0.0, 0.0, 1.8]'
```

The -1 option force rostopic to publish the message only once, if you want to publish the message at a specific frequency you will use:

```
$ rostopic pub /turtle1/cmd_vel geometry_msgs/Twist -r 1 -- '[2.0, 0.0, 0.0]' '[0.0, 0.0, 1.8]'
```

Where the -r 1 option specify that the message will be published at 1hz frequency



MESSAGES (ALSO SERVICES)

Getting information about msg/srv files

rosmmsg <command> [msg/srv_file]

subcommands (among the others)

show	Display the fields in the msg/srv.
list	Display names of all msg/srv.
package	List all the msg/srv in a package.
packages	List all packages containing the msg/srv.

rosmmsg show Pose

rosmmsg package nav_msgs



DEALING WITH SERVICES

Calling services from command line and getting information:

rosservice <command> [other_commands]

subcommand (among the others)

list	Print information about active services.
node	Print name of node providing a service.
call	Call the service with the given args.
args	List the arguments of a service.
type	Print the service type.
find	Find services by service type

rosservice call /reset

rosservice type /reset

BAGS



bag: file format to store messages data

Used to test different algorithm with the exact same input and to debug a system when it's not monitorable at runtime

To record a bag use:

`rosbag record`

to record all the topics use:

`$ rosbag record -a`

to record only a subset of the topic use:

`$ rosbag record topic1 topic2 etc`

BAGS



To get info regarding a beg use the command:

```
$ rosbag info bag_name
```

To play a bag run:

```
$ rosbag play bag_name
```

remember that to run rosbag you need an active ros session (roscore should be on)

Always monitor your bag size, sometimes logging all the topics (if you are working with cameras) is not the best idea because you will produce more data/sec than your max disk writing speed.

CREATE THE ROS WORKSPACE

ROBOTICS



POLITECNICO
MILANO 1863



CREATING THE WORKSPACE

ROS uses a custom compiling environment called **Catkin**

cmake/make with specific flags

Requires a workspace with a specific structure

Easy to setup and “easy” to use

```
mkdir -p ~/robotics/src
```

```
cd ~/robotics/
```

```
catkin_make
```

```
echo "source ~/robotics/devel/setup.bash" >> ~/.bashrc
```

```
source ~/.bashrc
```

WORKSPACE STRUCTURE



Source space (/src):

contains the source code of catkin packages.

Subfolder of this are the ROS packages you want to add to your system

All your stuff goes here!

Build space (/build):

space where cmake is invoked to build the catkin packages

Not where
catkin_make
is invoked!

cmake and catkin keep their cache information and other intermediate files here

Devel space (/devel):

Space where built targets are placed prior to being installed

PACKAGE CREATION



Command to create a new package

```
catkin_create_pkg [package_name] [depend1] [depend2] [depend3]
```

Before running the script `cd` to your src directory, then:

```
catkin_create_pkg pub_sub std_msgs rospy roscpp
```

Important Notes

roscpp and rospy are client libraries to use C++ and Python

!!Before being able to do that you should have creates a ros_workspace!!

PACKAGE CREATION



cd to the new package, the script should have created:

- CMakeLists.txt
- package.xml
- include folder
- src folder

cd to your catkin workspace root to compile the new package, simply using **catkin_make**

EDITORS/ IDEs

ROBOTICS



POLITECNICO
MILANO 1863

ROSED



rosed is part of the rosbash suite

Allow the user to edit files using directly the package name, rather than typing the entire path

```
rosed [package_name] [filename]
```

```
rosed roscpp Logger.msg
```

The default editor is vim

You can edit the .bashrc file setting a more user friendly editor

IDEs



No official IDE by ROS

C++ editor with ROS specific plugins

On ROS wiki you can find guides on how to properly configure the plugins

<http://wiki.ros.org/IDEs>

Simply add some features like easier compiling and some debug tools



Roboware



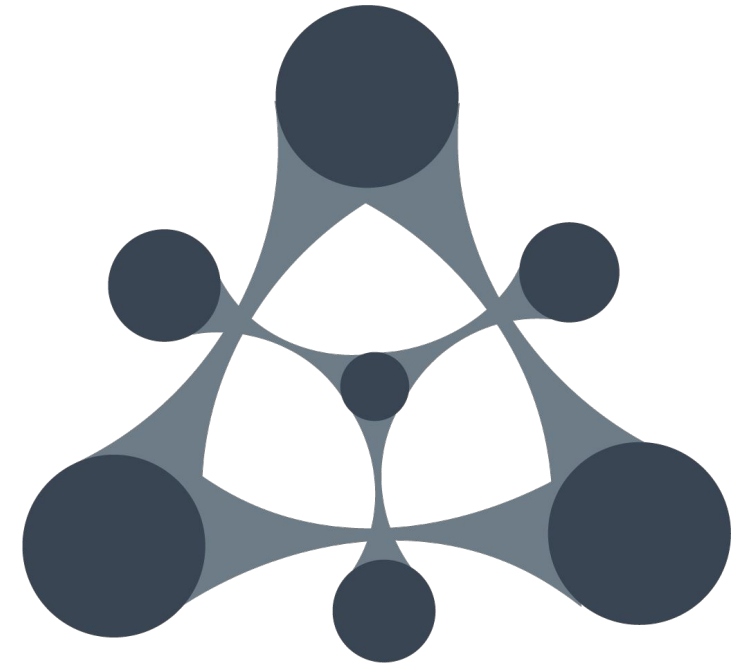
Based on Visual Studio

Designed for ROS

No need to install third parties plugin

Offers some functionalities:

- Run program directly inside Roboware
- Debugger
- Automatic file generation
- CMakeLists and Package.xml automatic update (partial)
- Integrated ros tool



ROS DEVELOPMENT

ROBOTICS



POLITECNICO
MILANO 1863

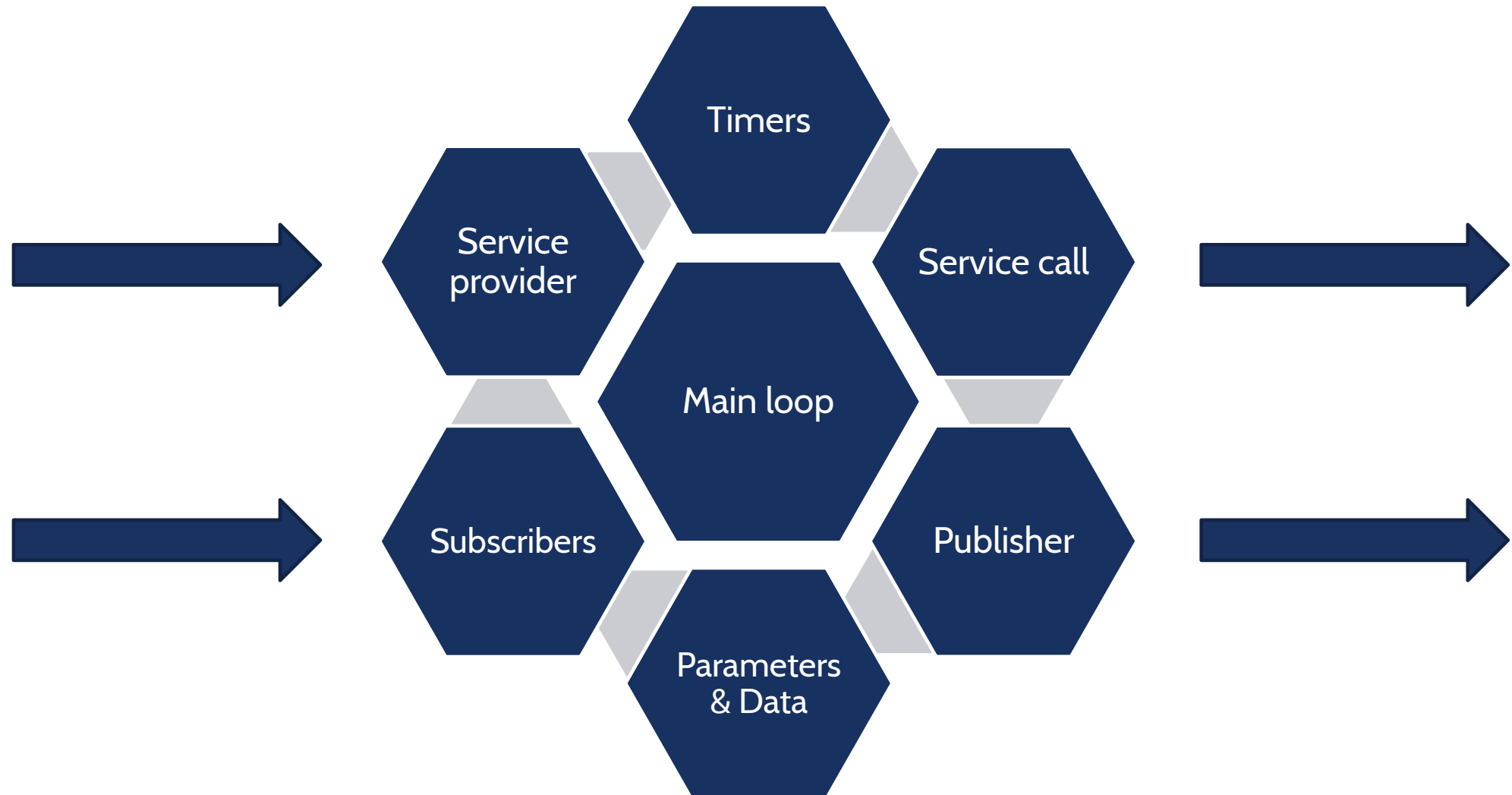
EVERYTHING HAPPENS IN NODES



Nodes are the main and atomic element of ROS. Each node is an independent process.

How do we create a node?
Write code in C++ or Python

INSIDE THE NODE





INITIALIZATION

Any node has to be registered to the ROS master using an unique identifier

The actual node is initialized using an handler

Each executable has an **unique name**

Each executable may have multiple handlers

```
void ros::init(argv, argc, std::string node_name, uint32_t options);  
ros::init(argc, argv, "my_node_name");  
ros::init(argc, argv, "my_node_name", ros::init_options::AnonymousName);  
  
ros::NodeHandle nh;
```

MAIN LOOP



Each ROS node loops waiting for something to do

At each loop checks:

- is there a message waiting to be received?

- is there a completed timer?

- is there a parameter to be reconfigured?

Two ways to implement the main loop:

- Automatically, no developer intervention

- Manual, specific sleep time and execution at each loop

```
ros::spin();
```

```
ros::Rate r(10); //10 hz
```

```
while (ros::ok()) {
```

```
    /* some execution */
```

```
    ros::spinOnce();
```

```
    r.sleep();
```

```
}
```

PARAMETERS



Stored in the parameter server and retrieved at the beginning of the execution

Adjustable at runtime using dynamic reconfigure

Global parameters and relative parameters (in the node namespace)

```
if(!nh.getParam("/global_name", global_name)) { /* :( */ }  
if(!nh.getParam("relative_name", relative_name)) { /* :( */ }  
nh.param<std::string>("param_name", default_param, "default_value");
```

PUBLISHER



Used to publish messages on a ROS topic

On declaration connect the publisher to a topic and define the type of the message

Can be called from everywhere

The frequency of the messages are not set

```
ros::Publisher pub = nh.advertise<std_msgs::String>("topic_name", 5);  
std_msgs::String str;  
str.data = "hello world";  
pub.publish(str);
```


SUBSCRIBER



Used to read messages from a ROS topic

On declaration connect the subscriber to a topic and define the type of the message

Call a specific function when receive a message

Operate at a given frequency

```
ros::Subscriber sub = nh.subscribe("topic_name", 10, callback);
```

```
void [class::]callback(const pack_name::msg_type::ConstPtr& msg)
```

TIMER



Used to execute something after a specific time (repeatable)

When the timer ends a callback function get called

Tied to ROS internal clock

```
ros::Timer timer = nh.createTimer(ros::Duration(0.5), callback);
```

```
void [class::]callback(const ros::TimerEvent& t)
```



SERVICE PROVIDER (SERVER)

Answer to a service call and execute some logic associated with the content of the call

On declaration connect to the callback with the implemented logic

The answer of the service is already in the callback

```
ros::ServiceServer s = nh.advertiseService("service", callback);
```

```
bool [class::]callback(pack::srv_type::Request& req, pack::srv_type::Response& res);
```



SERVICE PROVIDER (SERVER)

Generates the call for a specific service

On declaration is connected to the a service identified by a name

Can be called everywhere in the code

May result in a bad call

```
ros::ServiceClient cl = nh.serviceClient<pack::srv_type>("service");  
pack::srv_type srv;  
/* fill the service */  
if (cl.call(srv)) { /* :) */ } else { /* :( */ }
```



BUILDING YOUR CODE

```
cmake_minimum_required(VERSION 2.8.3)
project(package_name)
find_package(catkin REQUIRED COMPONENTS roscpp std_msgs genmsg)
add_message_files(FILES custom_message.msg)
add_service_files(FILES custom_service.srv)
generate_messages(DEPENDENCIES std_msgs)
catkin_package()

include_directories(include ${catkin_INCLUDE_DIRS})
add_executable(executable_name src/source_code.cpp)
target_link_libraries(executable_name ${catkin_LIBRARIES})
add_dependencies(executable_name package_name_generate_messages_cpp)
```



BUILDING YOUR CODE

```
cmake_minimum_required(VERSION 2.8.3)
project(package_name)
find_package(catkin REQUIRED COMPONENTS roscpp std_msgs genmsg)
add_message_files(FILES custom_message.msg)
add_service_files(FILES custom_service.srv)
generate_messages(DEPENDENCIES std_msgs)
catkin_package()
```

This is what you have to
change depending on
your code!

```
include_directories(include ${catkin_INCLUDE_DIRS})
add_executable(executable_name src/source_code.cpp)
target_link_libraries(executable_name ${catkin_LIBRARIES})
add_dependencies(executable_name package_name_generate_messages_cpp)
```



BUILDING YOUR CODE

```
cmake_minimum_required(VERSION 2.8.3)
project(package_name)
find_package(catkin REQUIRED COMPONENTS roscpp std_msgs genmsg)
add_message_files(FILES custom_message.msg)
add_service_files(FILES custom_service.srv)
generate_messages(DEPENDENCIES std_msgs)
catkin_package()
```

Only if you have custom
messages!

```
include_directories(include ${catkin_INCLUDE_DIRS})
add_executable(executable_name src/source_code.cpp)
target_link_libraries(executable_name ${catkin_LIBRARIES})
add_dependencies(executable_name package_name_generate_messages_cpp)
```