ROS 2 How To

Introduction:

- This is the website with EVERYTHING you need to know about ROS 2: https://docs.ros.org/en/iron/index.html. The information on this document is for further explanation of concepts that you will learn on the website. It is HIGHLY recommended that you follow the tutorials and how-to guides.
- First, you need actually to install ROS 2 on a Linux environment. Follow the instructions in the *Installation* section of the website. At the time this document is being written, we are using the Iron version of ROS 2. Make sure that the demo_nodes_py listener and demo_nodes_cpp talker work and can communicate with one another before continuing to the tutorials.
- You need to run *source /opt/ros/iron/setup.bash* to set up the ROS 2 environment or else ROS 2 commands won't work. This needs to be done for every new terminal window.

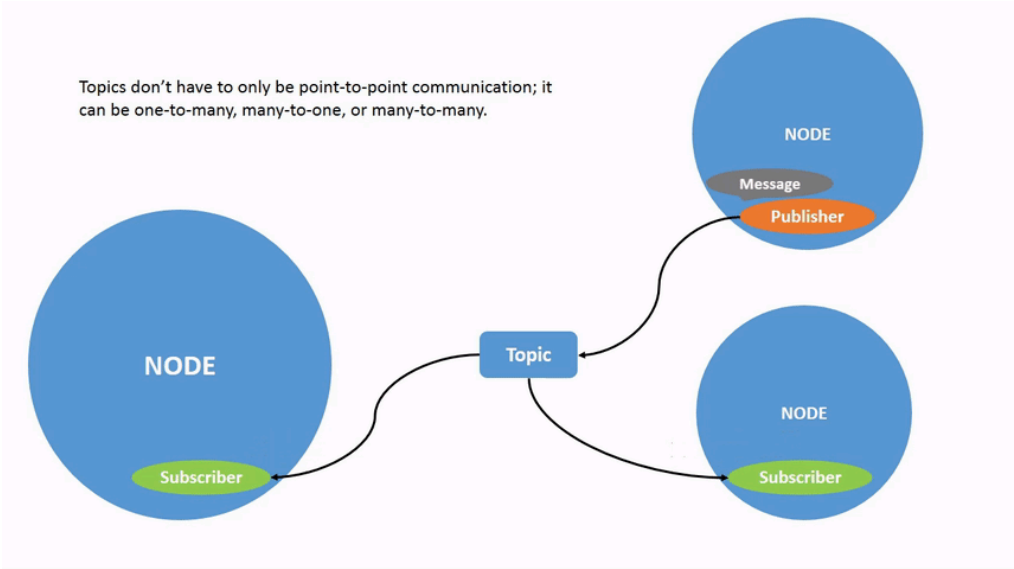
Nodes:

- A **node** is like a small unit of work or a functional block that helps in performing tasks within a robotic system. Think of it as a small program or module that can do specific things, like controlling a motor, processing sensor data, or performing calculations. Nodes communicate with each other by sending messages, allowing them to share information and work together to accomplish complex tasks.
- *ros2 node list*
    - shows all active nodes
- *ros2 run <node_name>. Ex: ros2 run turtlesim turtle_teleop_key*
    - Runs whatever node you want
- *ros2 run turtlesim turtlesim_node --ros-args --remap __node:=my_turtle*
    - This is for Remapping. Allows to reassign default node properties like node name, topic name, service name, etc., to custom values
    - This is useful when you need multiple versions of a node and want a way to keep track of which is which.
- *ros2 node info </node_name>. Ex: ros2 node info /my_turtle*
    - This will return a list of subscribers, publishers, services, and actions. These will be talked about more later.

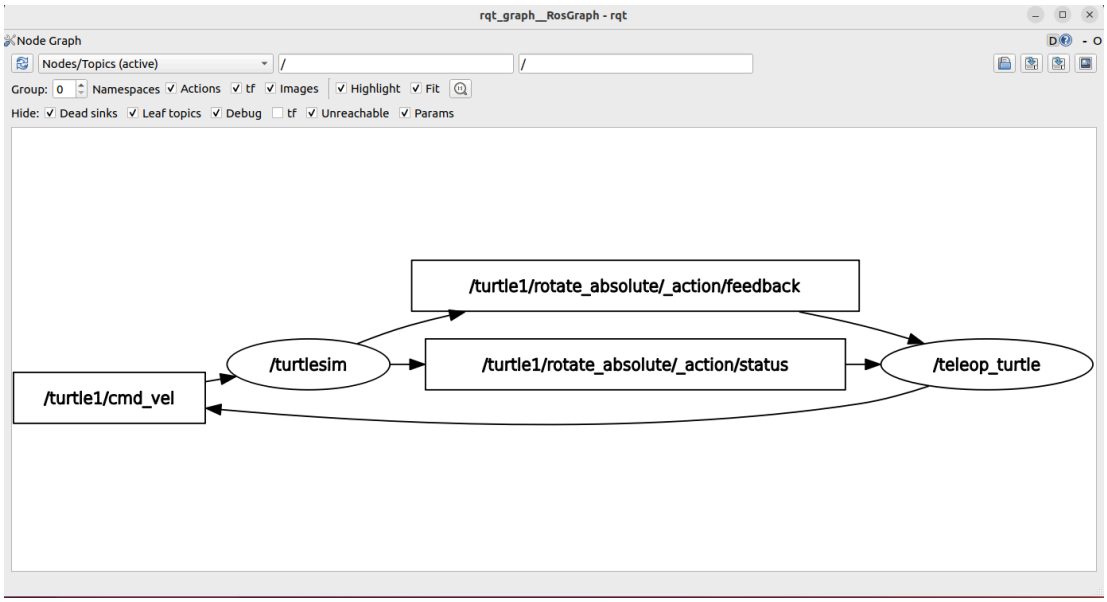
Topics & RQT Graph

- **Publishers** are in charge of sending data from a node to a topic. For example, a camera node in a robot might act as a publisher, continuously sending image data captured by the camera onto a topic named "/camera/image".

- **Topic**s are channels that connect publishers and subscribers. A topic essentially holds the data that a publisher sends out. Topics don't have to only be one-to-one communication; they can be one-to-many, many-to-one, or many-to-many.
- **Subscriber**s are entities that receive the data from topics. They perform actions based on specific data, like updating the position of a robot based on LiDAR data from a topic.
- **In summary**, publishers publish data onto topics, topics serve as communication channels for message passing, and subscribers receive data from topics. This decoupled communication mechanism enables flexible and modular design in ROS 2 systems, allowing components to interact seamlessly while maintaining loose coupling.



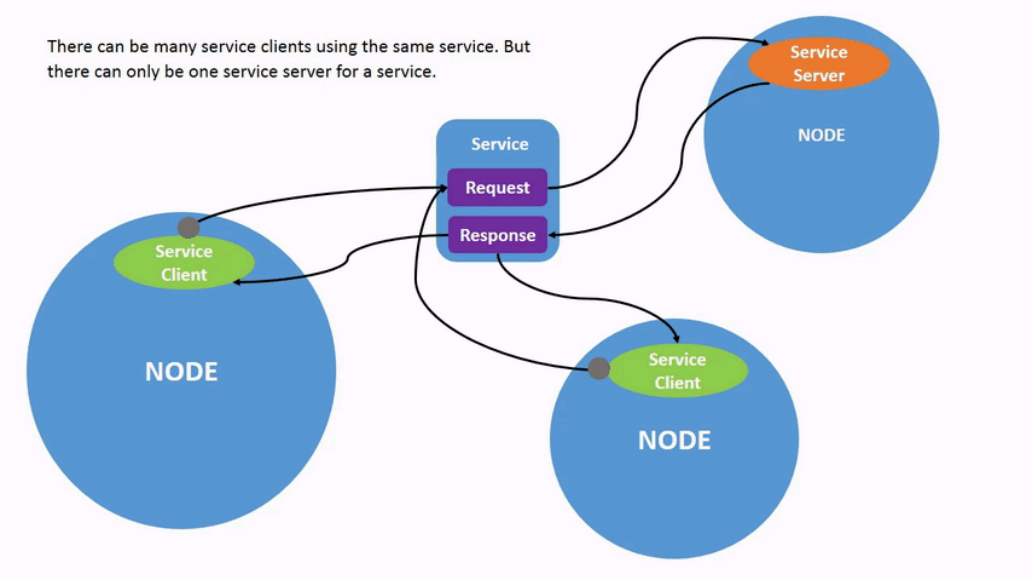
- *rqt_graph*
    - This command visualizes nodes, topics, and the connections between them



- In the example above, we have two nodes running (/turtlesim & /teleop_turtle). You can see that /teleop_turtle publishes information to the topic /cmd_vel which holds user input data that controls the turtle's movement.That topic then sends its data to the subscriber /turtlesim which updates the turtle's location.
- *ros2 topic echo <topic_name>. Ex: ros2 topic echo /turtle1/cmd_vel*
    - This command shows what data is being published to a topic. In the case of /cmd_vel, nothing will be shown because it is waiting for user input. When the turtle is moved around a bit, then it will show the data.
    - If you return to the rqt graph, it will show a new node that is subscribed to cmd_vel. This new node is the echo command.
- *ros2 topic info <topic_name>. Ex. ros2 topic info /turtle1/cmd_vel*
    - This command returns information like type, publisher count, and subscriber count
- *ros2 topic list -t*
    - Tells us what message type is used on each topic
    - For example, cmd_vel has a type geometry_msgs/msg/Twist
    - This means that in the package *geometry_msgs* there is a *msg* called *Twist*
- *ros2 interface show <type>. Ex: ros2 interface show geometry_msgs/msg/Twist*
    - Run this to learn the details about the type
- *ros2 topic pub <topic_name> <msg_type> '<args>'*
    - For publishing data on a topic directly
    - *args* is the actual data you want to send
    - *Ex: ros2 topic pub --once /turtle1/cmd_vel geometry_msgs/msg/Twist "{linear: {x: 2.0, y: 0.0, z: 0.0}, angular: {x: 0.0, y: 0.0, z: 1.8}}"*
    - The --*once* is optional and makes it only publish once
    - Replacing --*once* with --*rate 1* will publish the command in a steady stream at 1 Hz.
- *ros2 topic hz <topic_name>. Ex: ros2 topic hz /turtle1/pose*
    - Shows the rate at which the node is publishing to the topic

Services:

- **Services** are another method for nodes to communicate with one another. They are based on a call-and-response model versus the publisher-subscriber model of topics. While topics get **continual** updates, services only provide data when **specifically called by a client**.



- As we can see above, a node's service client sends a request message to a service. Then sends that request message to a node's service server. After the service server sends the response to the service which finally sends the response message to the service client.

- *ros2 service list*
  - Returns a list of all services active in the system
- *ros2 service type <service_name>*
  - Returns service type
  - *Ex: ros2 service type /clear* should return: std_srvs/srv/Empty
    - *Empty* type means the service call sends no data when making a request and receives no data when receiving a response
- *ros2 service list -t*
  - Returns all active services and their types
- *ros2 service find <type_name>*
  - Will return all services with a type
  - *Ex: ros2 service find std_srvs/srv/Empty* will return all services with type *Empty*

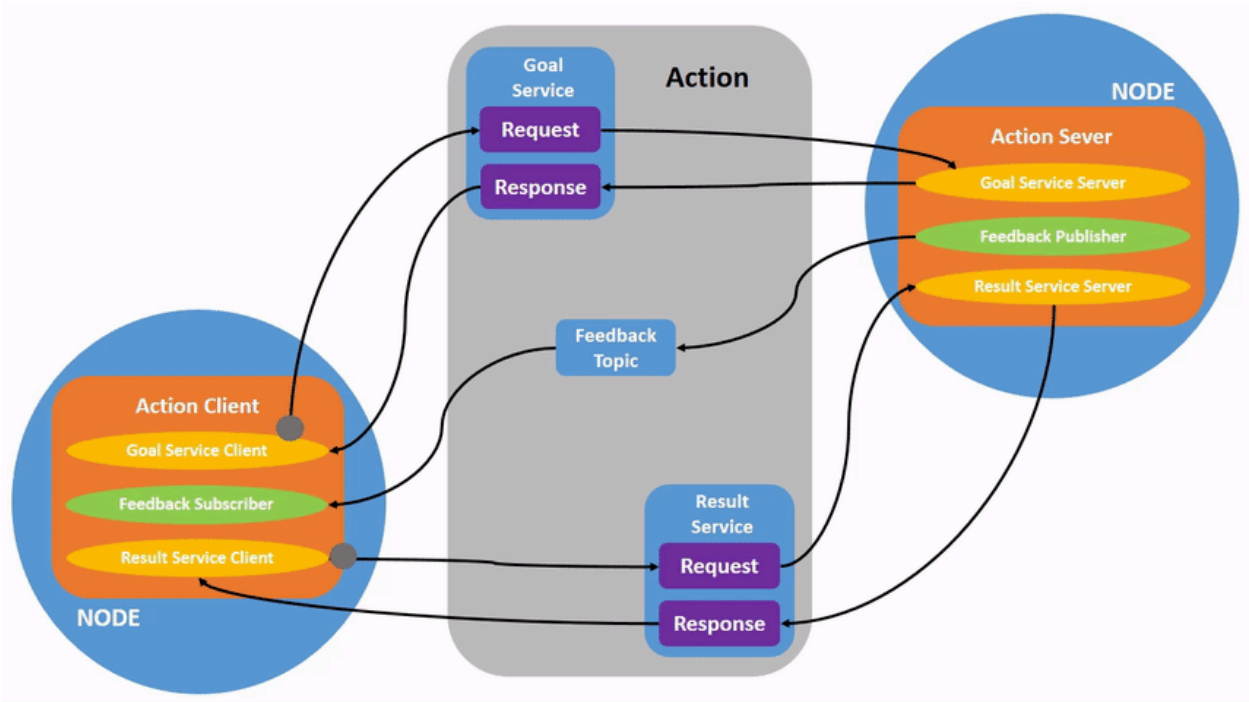
- *ros2 interface show <type_name>*
    - Returns the structure of the service
    - *Ex: ros2 interface show turtlesim/srv/Spawn will return*
      float32 x
      float32 y
      float32 theta
      string name # Optional. A unique name will be created and returned if this is empty
      ---
      string name
      Everything **above** "---" are request and everything **below** the "---" are responses
- *ros2 service call <service_name> <service_type> <arguments>*
    - Used for calling a service
    - The <arguments> part is optional. For example, you know that Empty type services don't have any arguments
    - *Ex: ros2 service call /clear std_srvs/srv/Empty* will clear the lines left behind by the turtle

Parameters

- A **parameter** is a configuration value of a node. It's like the node's settings.
- *ros2 param list*
    - Returns parameters belonging to your nodes
- *ros2 param get <node_name> <parameter_name>*
    - Returns the current value of the parameter
    - *Ex: ros2 param get /turtlesim background_g*
- *ros2 param set <node_name> <parameter_name> <value>*
    - Sets parameter value
    - *Ex: ros2 param set /turtlesim background_r 150*
- *ros2 param dump <node_name>*
    - Shows all nodes current parameters
    - *Ex: ros2 param dump /turtlesim*
- *ros2 param dump <node_name> > turtlesim.yaml*
    - This saves the current parameter values into a .yaml file in your current directory
    - You can access it (using nano) to view and edit the file
    - To load the file, use this command: ros2 param load <node_name> <parameter_file>
        - *Ex: ros2 param load /turtlesim turtlesim.yaml*
- *ros2 run <package_name> <executable_name> --ros-args --params-file <file_name>*
    - This will make the node startup with the specific parameters from the .yaml file
    - *Ex: ros2 run turtlesim turtlesim_node --ros-args --params-file turtlesim.yaml*

Actions

- **Actions** are one of the communication types in ROS 2 and are intended for long-running tasks. They consist of three parts: a goal, feedback, and a result.
    - Actions are built on topics and services
    - They function similarly to services, except actions can be canceled

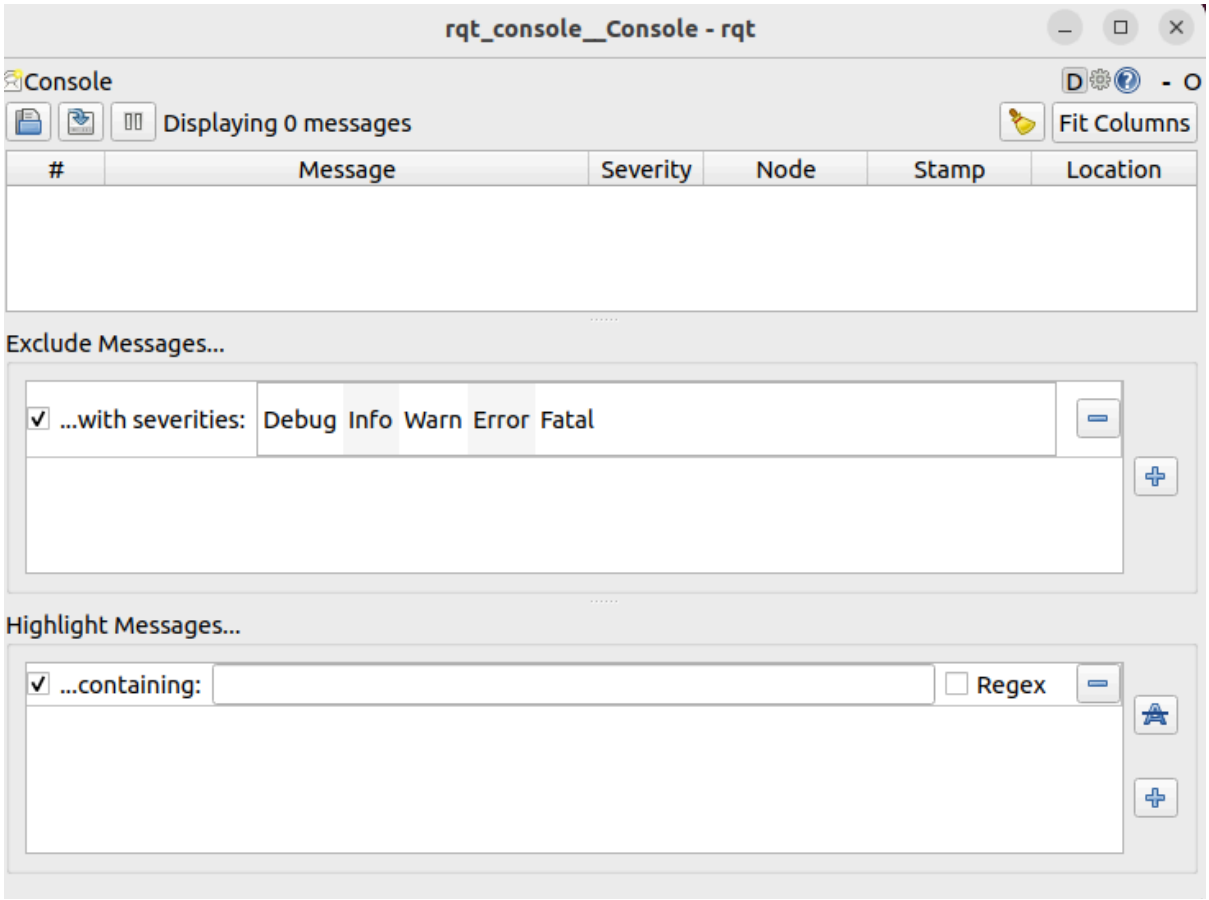


- *ros2 action list*
    - Lists all actions
- *ros2 action list -t*
    - Gives action types
- *ros2 action info <action_type>*
    - Gives information about the action
    - *Ex: ros2 action info /turtle1/rotate_absolute*
- *ros2 interface show <action_type>*
    - Gives structure of the action type
    - *Ex: ros2 interface show turtlesim/action/RotateAbsolute*
- *ros2 action send_goal <action_name> <action_type> <values>*
    - For sending an action goal from command line
    - *<values>* need to be in YAML format
    - *Ex: ros2 action send_goal /turtle1/rotate_absolute turtlesim/action/RotateAbsolute "{theta: 1.57}"*
    - Adding *--feedback* at the end of the command will show the progress of the action

Topic Vs Service Vs Action (When to use)

RQT Console

- **Rqt Console** is a GUI tool used to introspect log messages in ROS 2 in a more organized manner.



- Above is an example of the rqt_console
    - The first section is where log messages from the system will display
    - The middle section is for filtering messages by severity level. You can add more exclusion filters using the plus sign on the right of it
    - The bottom is for highlighting messages that include a string you input. You can add more filters as well.

Launching Nodes

- Using *ros2 launch* will launch all necessary nodes for the system instead of opening them one at a time with *ros2 run* on a new terminal for each.
- *ros2 launch turtlesim multisim.launch.py*
    - Here is a example that launches two turtlesim simulations
    - Below is the code for the.py file

```
# turtlesim/launch/multisim.launch.py

from launch import LaunchDescription
import launch_ros.actions

def generate_launch_description():
   return LaunchDescription([
      launch_ros.actions.Node(
         namespace= "turtlesim1", package='turtlesim', executable='turtlesim_node',
output='screen'),
      launch_ros.actions.Node(
         namespace= "turtlesim2", package='turtlesim', executable='turtlesim_node',
output='screen'),
   ])
```

Recording and Playing back data

- *ros2 bag* is a command line tool for recording data published on topics in your system. It accumulates the data passed on any number of topics and saves it in a database. You can then replay the data to reproduce the results of your tests and experiments. Recording topics is also a great way to share your work and allow others to recreate it.
- Note, it is wise to make a new directory (Ex: bag_files) before recording
- *ros2 bag record <topic_name>*
    - *Ex: ros2 bag record /turtle1/cmd_vel*
- You can record more than one topic with a single command and change the name of the bag file using *-o*. Below is an example of a bag file named *subset* and it is recording *cmd_vel* and *pose*
    - *ros2 bag record -o subset /turtle1/cmd_vel /turtle1/pose*
- *ros2 bag info <bag_file_name>*
    - This allows you to view information about the bag file
- *ros2 bag play <bag_file_name>*
    - This will replay the bag file

Colcon

- **Colcon** manages the development workflow of ROS projects, making it easier to build and test complex robotic systems.
- When you make a directory for your workspace, make a src file and put your packages in that directory
- *sudo apt install python3-colcon-common-extensions*
    - For installing colcon
- Colcon by default creates the following three directories:
    - The **build** directory will be where intermediate files are stored. For each package, a subfolder will be created in which e.g. CMake is being invoked.
    - The **install** directory is where each package will be installed to. By default each package will be installed into a separate subdirectory.
    - The **log** directory contains various logging information about each colcon invocation.
- After you git clone a ROS 2 repo, use *colcon build --symlink-install* and after its done use *colcon test* to run tests to make sure everything is working
- You will need to source the environment. Bash/bat files will be in your install directory that set the environment
    - *Ex: source install/setup.bash*

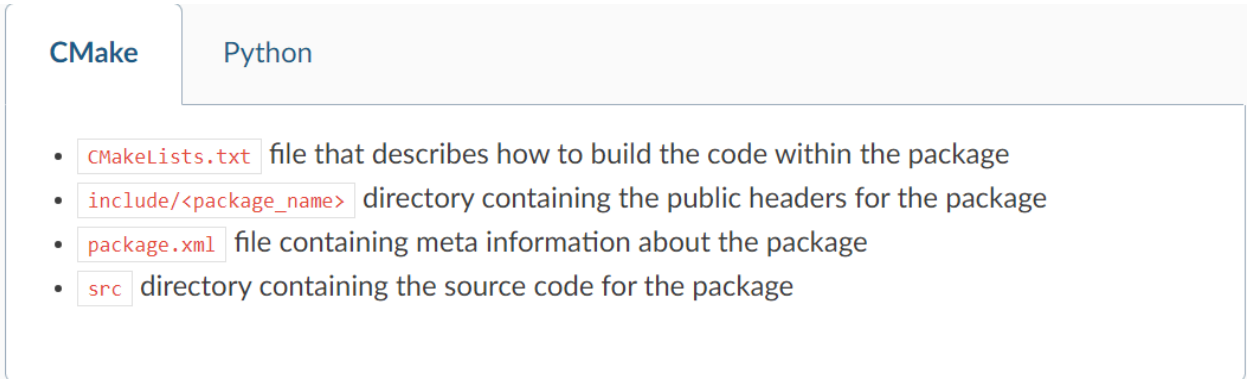
Workspace

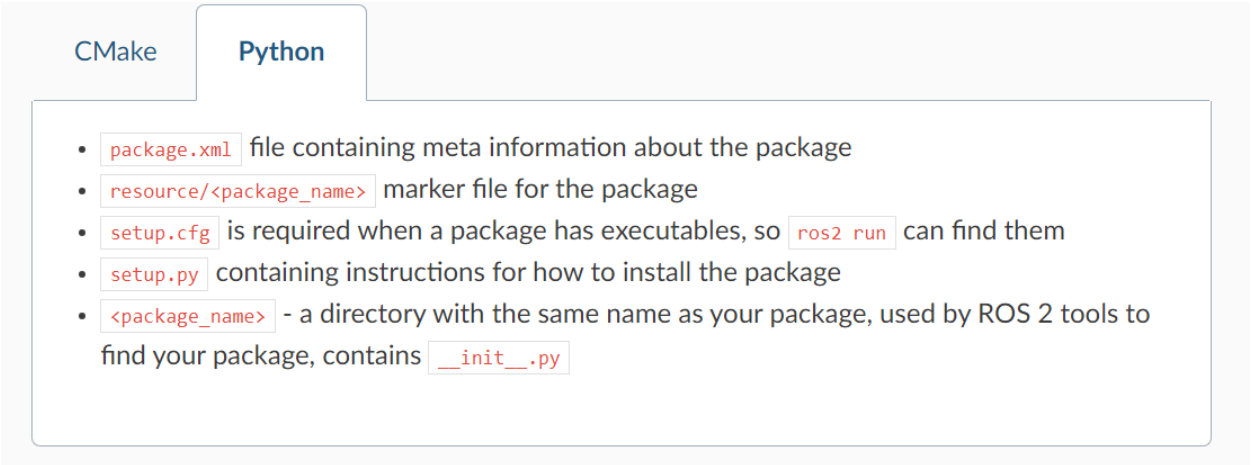
- The main ROS 2 installation will be the underlay for the tutorial
    - *source /opt/ros/iron/setup.bash*
- *rosdep install -i --from-path src --rosdistro iron -y*
    - Installs ROS 2 dependencies
- Other useful arguments for colcon build:
    - `--packages-up-to` builds the package you want, plus all its dependencies, but not the whole workspace (saves time)
    - `--symlink-install` saves you from having to rebuild every time you tweak python scripts
    - `--event-handlers console_direct+` shows console output while building (can otherwise be found in the log directory)
    - `--executor sequential` processes the packages one by one instead of using parallelism
- Overlay Vs Underlay
    - The **Underlay** is the base ROS 2 environment which holds the libraries, packages, and commands necessary for ROS2 to function

- The **Overlay** is a workspace that you create for a specific project
- For example, the *rqt_graph* command is part of the underlay, but a specific command that we develop for our workspace would be a part of the overlay.
- In the example from the website (Beginner:Client Libraries - Using a Workspace), we have turtlesim in both our underlay and overlay. But, when we run the turtlesim_node, the one from our overlay will run because the overlay has precedence over the contents of the underlay.

Packages

- A **package** is an organizational unit for ROS 2 code. If you want to be able to install your code or share it with others, then you'll need it organized in a package. With packages, you can release your ROS 2 work and allow others to build and use it easily.
- ROS 2 Python and CMake packages each have their own minimum required contents:





- These are some simple possible package structures:
- For CMake:
  my_package/
    CMakeLists.txt
    include/my_package/
    package.xml
    src/
- For Python
  my_package/
    package.xml
    resource/my_package
    setup.cfg
    setup.py
    my_package/

NOTE: Future students, please continue from where I left off. ROS2 has a lot of information and above will give you a great overview. But there are still things that are missing. I left off on the **Creating Packages section of Beginner: Client Libraries**