5^{th} Lecture
Robot Operating System Essentials – Summer School
Robot Operating System

IGMR Institute of Mechanism Theory, Machine Dynamics and Robotics | RWTH AACHEN UNIVERSITY

What is ROS (Robot Operating System)?

Paraphrased: „ROS is a set of **libraries** and **tools** to help build **robot applications**.

From drivers to state-of-the-art algorithms, it‘s all **open source**“

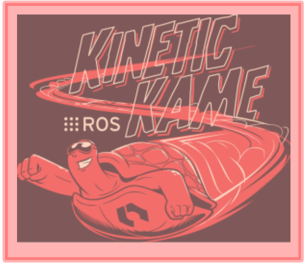
ROS as a framework provides:

- Drivers to talk to hardware
- Robot visualization and simulation
- Communication between heterogeneous systems
- Package management
- Open source libraries and packages

ROS distributions

As ROS is an actively managed, open source framework, it continually receives updates

ROS distributions are released regularly with alphabetically increasing names

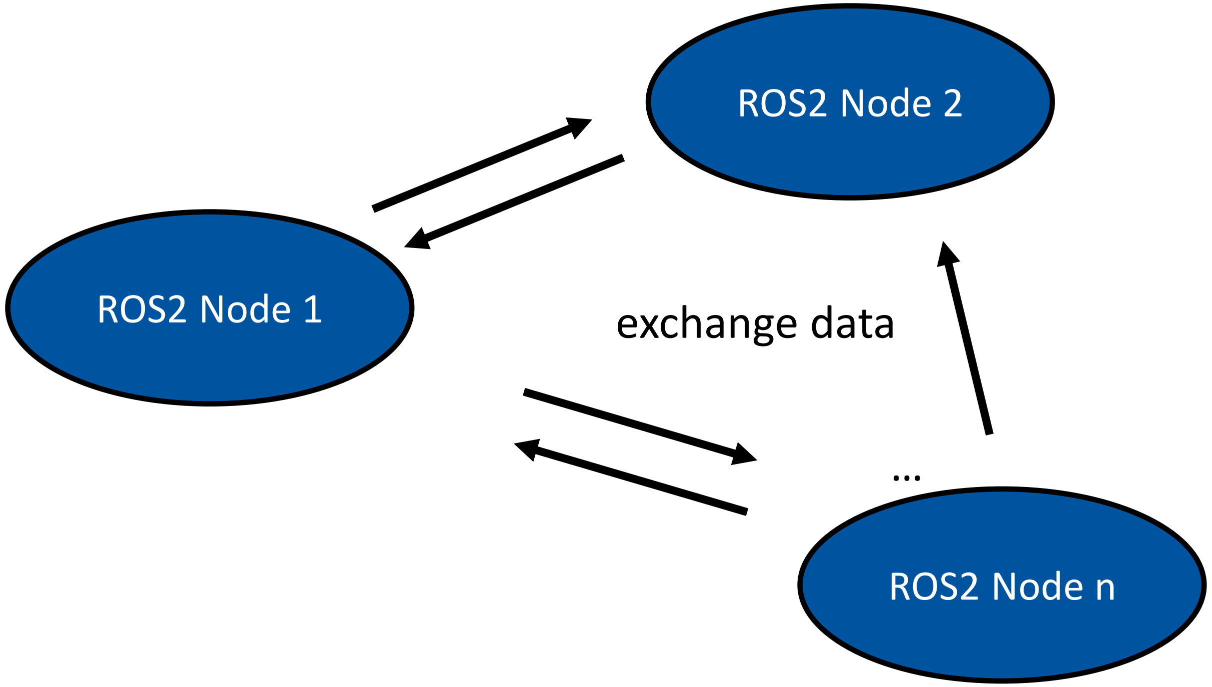


These distributions either have a short-term (1-2 years) or **long-term (5 year)** support

However, support for classical ROS (ROS1) has ended with ROS1 Noetic in 05.2025
Instead, the newer ROS2 distributions are/will be maintained actively – however, not everything has been fully ported yet
Knowing both ROS1 and ROS2 is useful, since both are in its core principles similar and allows understanding legacy code

ROS2 Framework: ROS2 Nodes

Running ROS programs are structured into separate executables (Nodes) who communicate decentrally with each other (via DDS)



The nodes (executables) simply contain code that is written in Python/C++

ROS2 Topics and ROS2 Messages

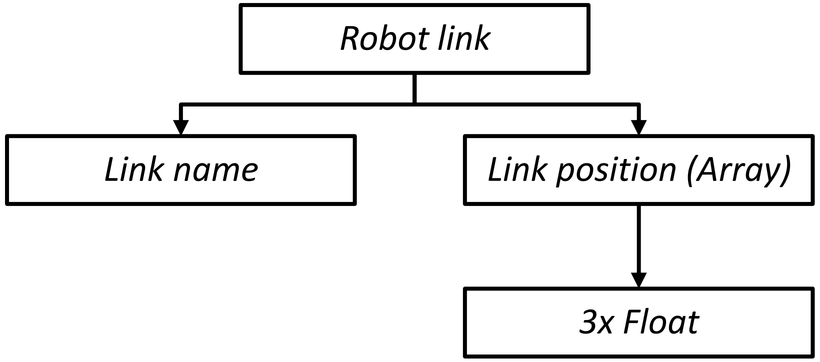
Communication between ROS2 nodes is organized in ROS2 Topics

A ROS2 Topic contains:
- A unique name
- An allowed message type
- ROS2 publishers
- ROS2 subscribers
- A history of published messages

ROS2 Topic

Name:	my_name
Message Type:	*String*
ROS2 publishers:	pub1, pub2, pub3
ROS2 subscribers:	sub1, sub2
Message log:	10:00 „Hi from pub1“
	10:01 „Hi from pub2“
	10:10 „Where is pub3?“
	10:10 „Hey, sorry for the delay“
	…

(Custom) *ROS2 messages* contain (in a dictionary):
- Primitive datatypes (int, string, double...) as key-value pairs
- Other ROS2 messages as key-value pairs
- Arrays with the above contents as key-value pairs



ROS2 publishers (in C++)

```
#include "std_msgs/msg/string.hpp“

class MyPublisher : public rclcpp:Node {
	public:
		rclcpp::Publisher<std_msgs::msg::String>::SharedPtr publisher_;
		int msg_queue_length = 1000;
		rclcpp::TimerBase::SharedPtr timer_;

		MyPublisher():Node(“node_name”) {
			publisher_ = this->create_publisher<std_msgs::msg::String>("topic_name", msg_queue_length);
			timer_ = this->create_wall_timer(std::chrono::seconds(1), std::bind(&MyPublisher::pub, this));
		}

		void pub() {
			auto msg = std_msgs::msg::String();
			msg.data = ...
			publisher_->publish(msg);
		}
}

int main(int argc, char **argv) {
	rclcpp::init(argc, argv);
	rclcpp::spin(std::make_shared<MyPublisher>());
}
```

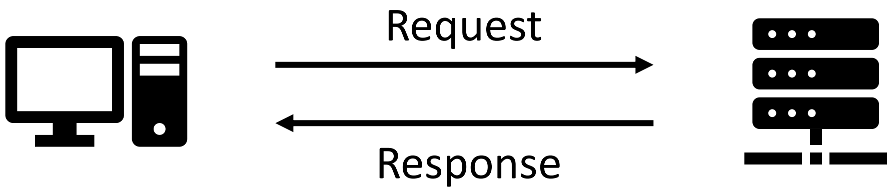

IGMR Institute of Mechanism Theory, Machine Dynamics and Robotics
RWTH AACHEN UNIVERSITY

ROS2 subscribers (in C++)

```
#include "std_msgs/msg/string.hpp"

class MySubscriber : public rclcpp:Node {
	public:
		rclcpp::Subscription<std_msgs::msg::String>::SharedPtr subscriber_;
		int msg_queue_length = 1000;

		MySubscriber():Node(“node_name”) {
			subscriber _ = this->create_subscription<std_msgs::msg::String>("topic_name",
						msg_queue_length, std::bind(&MySubscriber::sub, this, _1));
		}

		void sub() {
			do_something();
		}
}

int main(int argc, char **argv) {
	rclcpp::init(argc, argv);
	rclcpp::spin(std::make_shared<MySubscriber>());
}
```

ROS2 services

When communication is not *push-based* (publisher broadcasts messages, subscribers listen) but *pull-based*, **ROS2 services** allow for *clients* to send **requests** to a *server*



A service requires the server to define the format in **.srv files** containing:

- service name
- service inputs (ROS2 messages)
- service outpus (ROS2 messages)

ROS2 Service	
Name:	icecream_service
Service Inputs:	cone or cup
	list of flavors
Service Outputs:	icecream

The concept of ROS2 services realizes more structured communication

When defining services, in the background these are implemented/auto-generated with nothing more than corresponding ROS2 topics…

ROS2 servers (in C++)

```
#include "my_package/IceCream.h"

void add(const std::shared_ptr<my_package::srv::IceCream::Request> request,
                std::shared_ptr<my_package::srv::IceCream> response) {
        if (req.cone) {
                // Proceed with cone
        } else {
                // Proceed with cup
        }
        // Make icecream
}

int main(int argc, char **argv) {
        rclcpp::init(argc, argv);
        std::shared_ptr<rclcpp::Node> node = rclcpp::Node::make_shared("node_name");
        rclcpp::Service<my_package::srv::IceCream>::SharedPtr service =
                node->create_service<my_package::srv::IceCream>("service_name", & serviceIceCream);
        rclcpp::spin(node);
}
```

ROS2 clients (in C++)

```
#include “my_package/IceCream.h"

int main(int argc, char **argv) {
        rclcpp::init(argc, argv);
        std::shared_ptr<rclcpp::Node> node = rclcpp::Node::make_shared("node_name");
        rclcpp::Client<my_package::srv::IceCream>::SharedPtr client =
                node->create_client<my_package::srv::IceCream>("service_name");
        auto request = std::make_shared<my_package::srv::IceCream::Request>();
        request->cone = true;
        request->flavors = {vanilla, chocolate, strawberry};

        auto result = client->async_send_request(request);
        if (result.valid) {
                // Service successful
        } else {
                // Service unsuccessful (no such flavor/out of stock etc.)
        }
}
```

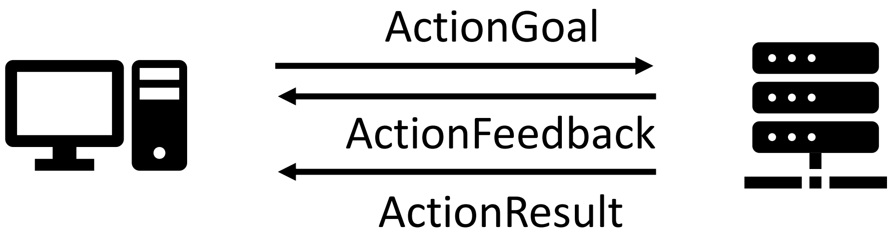
ROS2 actions

Using ROS2 services involves **two messages**: 1. Service request 2. Service response
This type of communication is useful when the **time** between request, execution and response is **short**

For longer processes however, one might wish to:

- Cancel the request
- Get periodic feedback during execution

This is realized with ROS2 actions



A valid ROS2 action server must define action messages composed of:

- Goal: The „service“ that the client requests
- Feedback: Periodic reponse from the action server
- Result: A response that the ROS2 action server sends **once** at the end of the action

ROS2 Action	
Name:	my_progressbar
Action Goal:	download_url
Action Feedback:	progress
Action Result:	.exe

In the background, ROS2 actions too are realized using appropriate ROS2 topics…

ROS2 action servers (in C++)

```
#include “my_package/action/myAction.hpp“

class MyActionServer : public rclcpp:Node {
	public:
		using myAction = mypackage::action::MyAction;
		using goalHandle = rclcpp_action::ServerGoalHandle<myAction>;
		rclcpp_action::Server<myAction>::SharedPtr action_server_;
		explicit MyActionServer(const rclcpp::NodeOptions& options = rclcpp::NodeOptions()) : Node(“node_name", options) {
			this->action_server_ = rclcpp_action::create_server<myAction>(
				this,
				“actionTopic",
				[this](const rclcpp_action::GoalUUID& uuid, std::shared_ptr<const myAction::Goal> goal) {return this->handle_goal(uuid, goal);},
				[this](const std::shared_ptr<goalHandle> goal_handle) {return this->handle_cancel(goal_handle);},
				[this](const std::shared_ptr<goalHandle> goal_handle) {this->handle_accepted(goal_handle);}
				);
		}
		rclcpp_action::GoalResponse handle_goal(const rclcpp_action::GoalUUID & uuid, std::shared_ptr<const myAction::Goal> goal);
		rclcpp_action::CancelResponse handle_cancel(const std::shared_ptr<goalHandle> goal_handle);
		void handle_accepted(const std::shared_ptr<goalHandle> goal_handle);
		void execute(const std::shared_ptr<goalHandle> goal_handle) {
			if (goal_handle->is_canceling()) {
				goal_handle->canceled(result);
			} else {
				goal_handle->publish_feedback(feedback);
			}
			goal_handle->succeed(result);
		}
}

RCLCPP_COMPONENTS_REGISTER_NODE(MyActionServer)
```

ROS2 action clients (in C++)

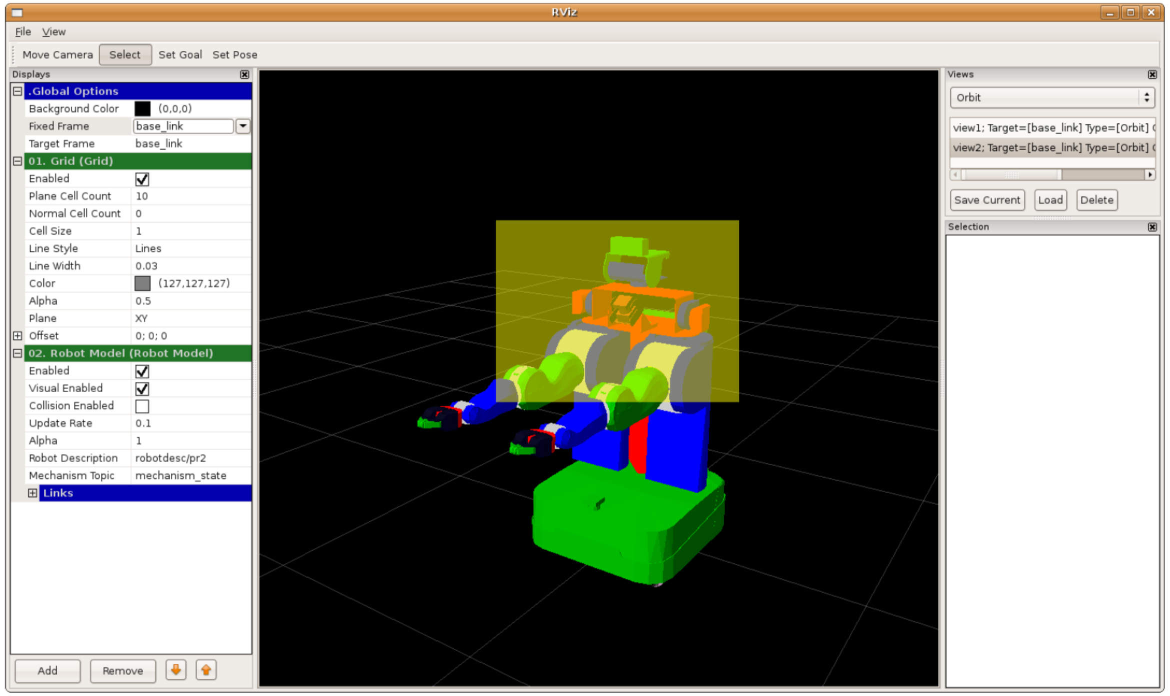
```
#include “my_package/action/myAction.hpp“

class MyActionClient : public rclcpp:Node {
		public:
				using myAction = mypackage::action::MyAction;
				using goalHandle = rclcpp_action::ServerGoalHandle<myAction>;
				rclcpp_action::Client<myAction>::SharedPtr action_client_;
				explicit MyActionClient(const rclcpp::NodeOptions& options = rclcpp::NodeOptions()) : Node(“node_name", options) {
						this->action_client_ = rclcpp_action::create_client<myAction>(
								this,
								“actionTopic");
				}
				send_goal();
				goal_response_cb();
				feedback_cb();
				result_cb();
}

RCLCPP_COMPONENTS_REGISTER_NODE(MyActionClient)
```

ROS2 visualizations: RViz2

RViz2 (ROS2 visualization) is a software for visualizing the contents that are published on ROS2 Topics

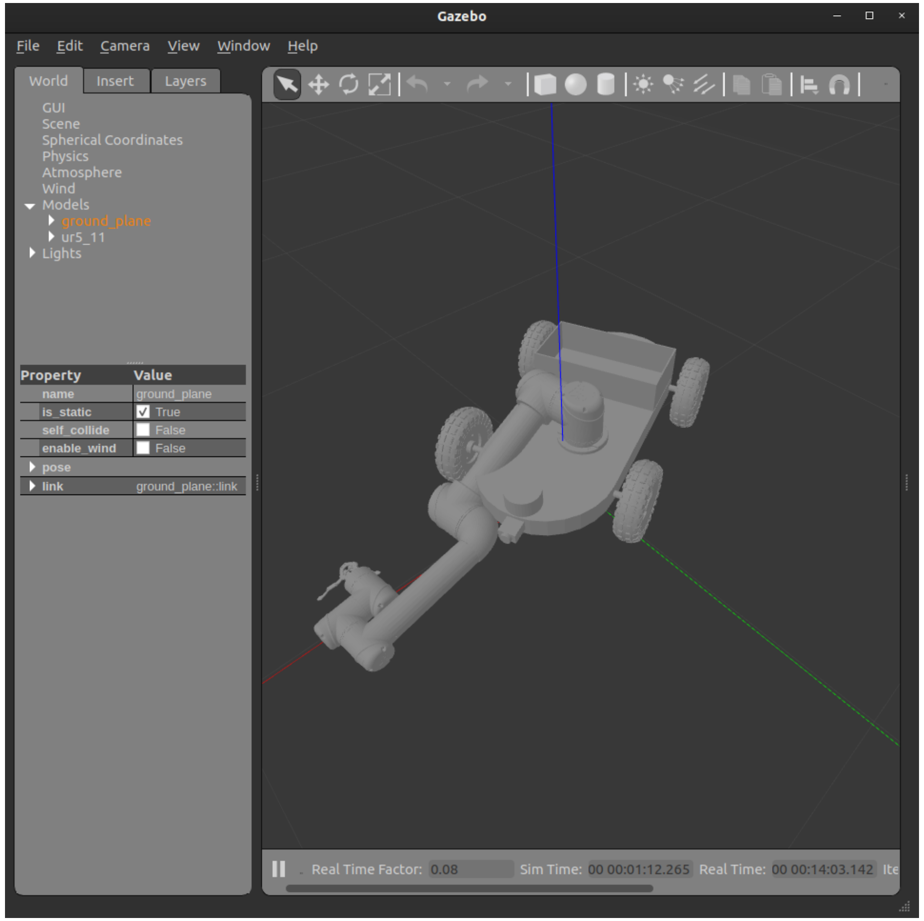
This includes (but is not limited to):

- (Robot) models (primitive shapes/meshes…)
- (Camera) video feed
- Pointcloud data
- Robot trajectories
- Custom markers (shapes, arrows…)
- A rudimentary graphical user interface



ROS2 simulations: Gazebo

Gazebo is a robot simulator that is frequently used in combination with ROS2



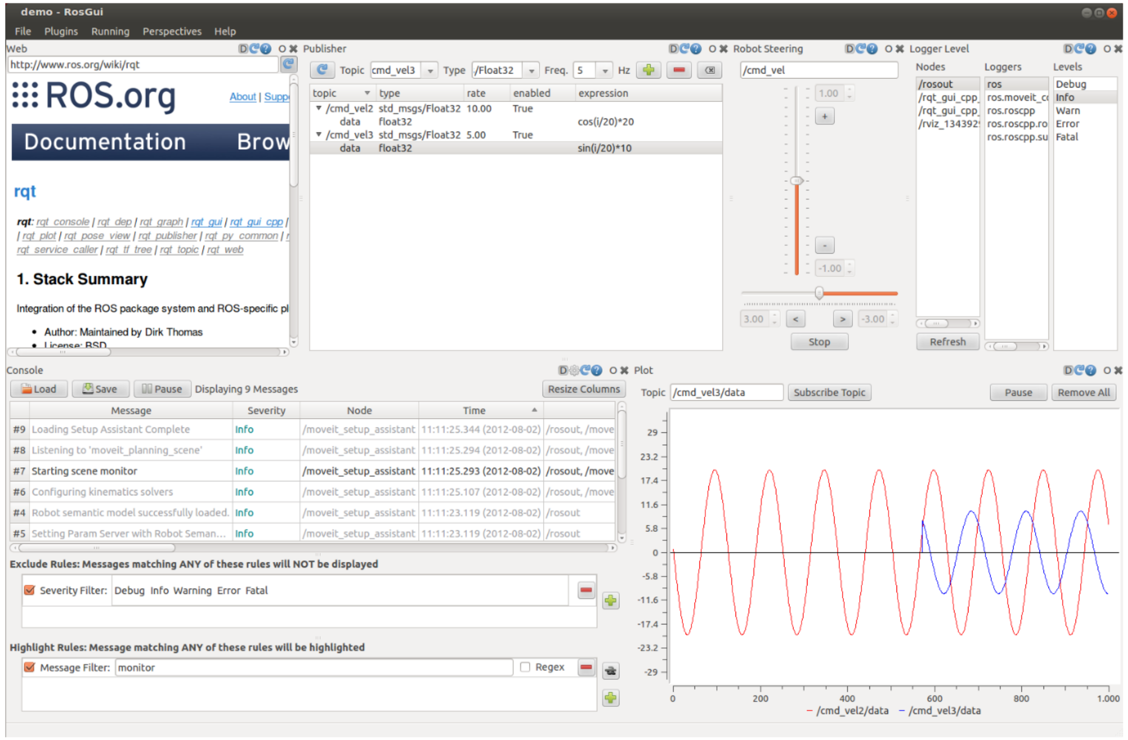
As a simulator, it is different to RViz and supports:

- Physical simulation of bodies
- Robot simulation
- Simulation of sensors (cameras, LIDARs, odometry etc.)
- Simulation of lighting
- Animations (simulation without physics)
- (Plugins) to allow communication with the outside world (ROS2)

ROS2 visualizations: RQT

As ROS2 is Linux-based, many operations are commony done in the terminal

For debugging purposes, a graphical user interface may be at times very helpful



As such, RQT can be used for:

- ROS2 topic visualization
- ROS2 nodes visualization
- Calling ROS2 services
- Mathematical plotting
- Basic GUI element interactions

Catkin workspaces and ROS2 packages

When installing ROS2, the software skeleton and a basic set of libraries is installed to /opt/ros/<version>/

When writing your own code, it is structured in **colcon workspaces**

A colcon workspace becomes *valid* as soon as it contains a **/src** folder
When building the code, ROS2 *automatically creates* a **/build** and a **/install** folder

Specifically, the Python/C++ code is located in **ROS2 packages**
which are located in the **/src** folder

When running ROS2 programs, the code dependencies may span the **/src** folder
of the colcon workspace (with all the contained **ROS2 packages**)

A **ROS2 package** is only valid once it contains a CMakeLists.txt and package.xml

colcon workspaces

```
my_workspace
  build (created automatically)
  install (created automatically)
    setup.bash
  src
    my_package1
      CMakeLists.txt
      package.xml
      msg
      launch
      include
      src
    my_package2
    my_package3
```

package.xml

Upon creating each **ROS2 package**, ROS2 automatically generates a unique CMakeLists.txt file and package.xml file

The package.xml file contains data such as package name, version numbers, authors, maintainers and **dependencies on other colcon packages**

The most **relevant types of dependencies** are:

- Build Dependencies: Specify which packages are needed to build this package
- Build Export Dependencies: Specify which packages are needed to build libraries against this package
- Execution Dependencies: Specify which packages are needed to run code in this package
- Build Tool Dependencies: (Like build dependencies, but on a meta-level => often 1-2 packages here)

CMakeLists.txt

Upon creating each **ROS2 package**, ROS2 automatically generates a unique CMakeLists.txt file and package.xml file

The CMakeLists.txt file is the **input to the CMake build system** for building software packages
(applies to **both Python ROS2 projects** and **C++ ROS2 projects**)

As such, it is used to define build dependencies, ROS2 message/service/action generation, executable/library definitions etc.

- find_package(): List other **CMake** or **Catkin packages** needed for building this ROS2 package
- rosidl_generate_interfaces(): List files containing **custom ROS2 messages, servies and actions** to build
- include_directories(): List locations/files of this package with headers to include
- add_library(): Specify **libraries** (used by other libraries and executables) to build
- add_executable(): Specify **exectuables (nodes)** to build
- target_link_libraries(): Define for **each executable** which **libraries it depends** on
- ament_target_dependencies(): Like target_link_libraries, but ROS2 (ament) specific

Important commands and shortcuts in when using ROS2

Important commands:

ros2 run	run ROS2 executables (nodes)
ros2 launch	run ROS2 launch files (usually multiple nodes)
colcon build	build all the packages in the /src folder of the current workspace
source install/setup.bash OR **. install/setup.bash**	(only for current terminal) the bash file of the current workspace
ros2 topic <…> (list, info, echo)	list ros2 topics, get info on one topic, get contents of one topic
ros2 service <…> (list, info, echo)	list ros2 services, get info on one service, get contents of one service
rviz2	start rviz
rqt	start rqt
cd my/relative/path and cd ..	change directory to my/relative/path or change back
apt-get install ros-<version>-<pkg-name>	(ex. ros-humble-rosserial)

Important shortcuts:

Ctrl+c	stop process in terminal (exit gedit, stop ROS2 node, stop ROS2 launch…)
Tabulator	autocomplete commands and paths (always use this!)
Ctrl+r	reverse search in command-history
Keyarrow up/down	cycle back/forth in command history
Ctrl+Shift+c	copy marked text from a terminal
Ctrl+Shift+v	paste text into a terminal