Tutorial 1

- Tutorial 1: https://www.youtube.com/watch?v=EU-QaO6xTv4
- Written by Milad Abdi

Docker

- Allows for easy deployment of software in a loosely isolated sandbox (containers).
- Can run many containers simultaneously.
- Containers are **lightweight** and contain everything needed to run the application.
  - This fixes "but it worked on my machine" problems.
- Allows us to package an application with **all its dependencies into a standardized unit**.
- **Architecture**:
  - Client: You can use docker on the command line (terminal) or use Docker Desktop.
  - Serverside: Handles all containers and images.
    - Registry (DockerHub): Place for publicly available containers and images.
- **Images**:
  - Read-only template with instructions for creating a Docker container. (Instructions to create your sandbox with whatever software you want installed).
  - Image often based on another image. E.g., Your ROS 2 image would have an image of Ubuntu with ROS 2 added onto it.
  - You can create your own images or use images made by others.
- **Building Images**:
  - Create `DockerFile` and write the instructions syntax which define the steps needed to **create the image** and **run it**.
  - **Layer caching**: Each instruction creates a layer in the image. If you change your Dockerfile, only the changed layers will be rebuilt. Thus, Docker is lightweight and fast.
- **Containers**:
  - **Runnable instance of an image** (you can have multiple instances of the same container at once).
    - You can create, start, stop, move, or delete a container.
    - You can connect a container to one or more networks (to let them to talk to each other or talk to you separately), and attach storage to it.
  - Container is **well isolated** from other containers and host machine. You can control how much isolation between your container and other containers or host machine.
  - Container is **defined** by **its image** and the **configuration options** you give it when you **create or start** the container.

- Container removed -> Any changes to its state that are **not** stored in persistent storage disappear.

Docker Commands

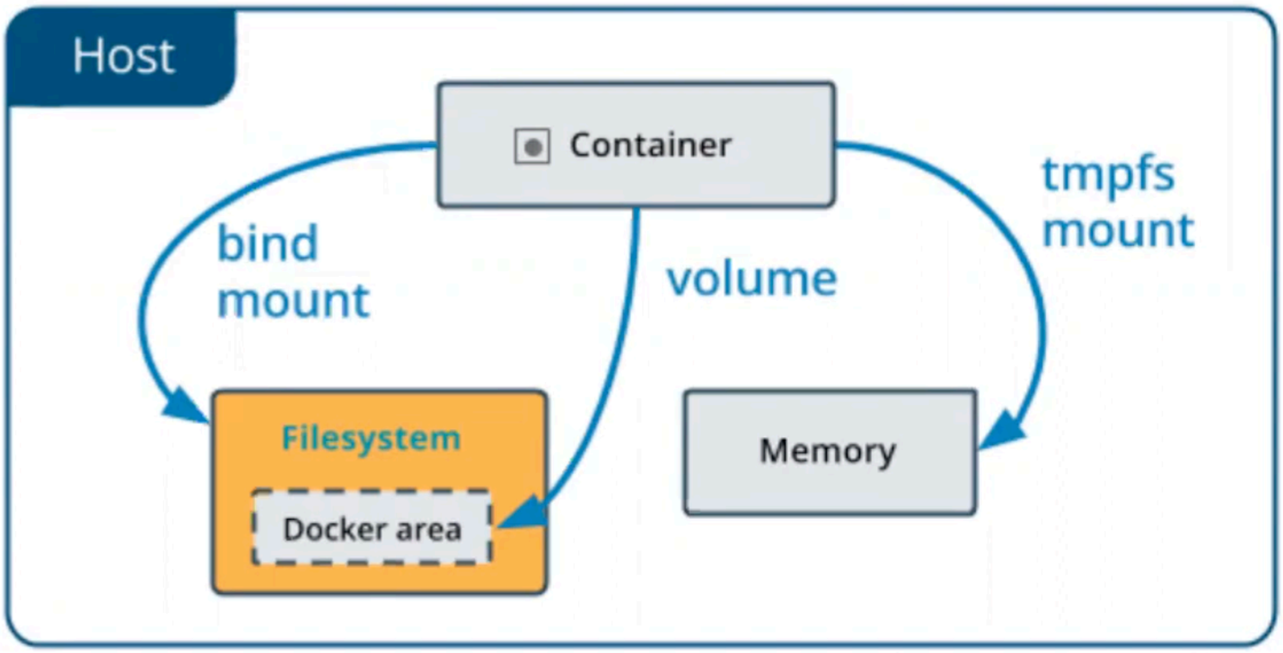
- **Download an image from a registry**: `docker image pull [OPTIONS] NAME[:TAG|@DIGEST]` .
- **Create and run a new container from image**: `docker run [OPTIONS] IMAGE [COMMAND] [ARG...]` .
  - `--name` : Assign name to container
  - `-it` : Combines `--interactive (-i)` and `--tty(-t)` : Keep STDIN open even if not attached, and allocate pseudo-TTY (interface)
  - `-v` : Bind mount a volume
  - `--rm` : Auto remove container when it exists
  - `--net/--network` : Connects a container to a network
  - `-p` : Publish a container's port(s) to the host
- **Build an image from a Dockerfile. A build's context is the set of files in the specified `PATH` or `URL`**: `docker build [OPTIONS] PATH | URL | -` .
  - `-f` : Name of the Dockerfile, default is 'Path/Dockerfile'
  - `-force-rm` : Always remove intermediate containers
  - `-no-cache` : Do not use cache when building the image.
- **List containers**: `docker ps [OPTIONS]` .
  - `-a` : Show all containers, running and stopped.
  - `-s` : Display total file size.
- **List images**: `docker images [OPTIONS] [REPOSITORY[:TAG]]` .
  - `-a` : Show all images, default hides intermediate images.
- **Remove one or more containers**: `docker rm [OPTIONS] CONTAINER [CONTAINER...]` .
  - `-f` : Force the removal of a running container (uses SIGKILL)
- **Remove one or more images**: `docker rmi [OPTIONS] IMAGE [IMAGE...]` .
  - `-f` : Force removal of the image.
- **Execute a command in a running container (usually to open an interactive bash in a running container)**: `docker exec [OPTIONS] CONTAINER COMMAND [ARG...]` .
  - `-it` : Combines `--interactive (-i)` and `--tty(-t)` : Keep STDIN open even if not attached, and allocate pseudo-TTY
- **Copy folders/files between a container and the local filesystem (behaves like UNIX `cp`, having similar options for copying recursively, etc)**:
  - `docker cp [OPTIONS] CONTAINER:SRC_PATH DEST_PATH` or
  - `docker cp [OPTIONS] SRC_PATH CONTAINER:DEST_PATH` .

Dockerfile

- **Dockerfile**: Dockerfiles are instructions for Docker to build images automatically. Using `docker build` users can create an automated build that executes several command-line instructions in succession (**in order from top to bottom**).
- General Syntax Format:
    - `# Comment`
    - `INSTRUCTION arguments`
- `FROM`: Create a new build stage from a base image:
    - Dockerfile must begin with a `FROM` instruction (may only be preceded by `ARG` instructions, which declare arguments that are used in `FROM` lines).
    - `FROM` instruction specifies the Parent Image from which you are building. E.g., `FROM ubuntu:24.04`.
- `RUN`: Executes build commands. Has two forms:
    1. Shell form: `RUN <command>`: command is run in shell.
        - Default shell is `/bin/sh -c` on linux, or `cmd /S /C` on windows.
    2. Exec form: `RUN ["executable", "param1", "param2"]`.
    - `git` dependencies, the directory you go into after `cd` when called by `RUN` **do not persist**. Only modifications to the file system will persist.
- `CMD`: Sets the command to be executed when running a container from an image. Has three forms:
    1. `CMD ["executable","param1","param2"]` (exec form, preferred)
    2. `CMD ["param1","param2"]` (exec form, as default parameters to `ENTRYPOINT`)
    3. `CMD command param1 param2` (shell form)
    - There can only be one `CMD` instruction in a Dockerfile. If you list more than one `CMD`, only the last one takes effect.
    - The purpose of a `CMD` is to provide defaults for an executing container. These defaults can include an executable, or they can omit the executable, in which case you must specify an `ENTRYPOINT` instruction as well.
- `ENV <key>=<value> [<key>=<value>...]`: Set environment variables.
    - Then `ENV` instruction sets the environment variable `<key>` to the value `<value>`. This value will be interpreted for other environment variables.
    - The environment variables set using `ENV` **will persist** when a container is run from the resulting image.
- `ARG <key>=<value> [<key>=<value>...`: Use build-time variables.
    - Unlike `ENV`, `ARG` **will not persist** when a container is run from the resulting image.
- `COPY`: Copies new files or directories from `<src>` and adds them to the filesystem of the image at the path `<dest>`. Has two forms:

   1. `COPY [--chown=<user>:<group>] <src>... <dest>`
   2. `COPY [--chown=<user>:<group>] ["<src>", ... "<dest>"]`
      - `[--chown=<user>:<group>]` is an `[OPTION]` that is used to change the user and group ownership of files or directories.
- `ENTYPOINT` : Allows you to configure a container that will run as an executable. Has 2 forms:
   1. `ENTRYPOINT ["executable", "param1", "param2"]` (exec form, preferred)
   2. `ENTRYPOINT command param1 param2` (shell form)
- `WORKDIR /path/to/workdir` : Specifies the working directory.
   - Becomes the directory you'll be in when starting an interactive bash session.
   - If done before any of `RUN` , `CMD` , `ENTRYPOINT` , `COPY` and `ADD` instructions, the work directory for them will be the path you give instead of `/` by default.
- To learn more about these commands: https://docs.docker.com/reference/dockerfile/

Bind Mounts and Volume



- **Bind Mount**: Mounts an **existing file or directory** on the host machine into a container. The file or directory is referenced by its absolute path on the host machine.
- **Volume**: Creates a **new directory within Docker's storage directory** on the host machine, and Docker manages that directory's content.
- **Tmpfs (Temporary file system) mount**: Shares memory between container and host system.
- **Starting a container with a bind mount**. Two ways:
   1. `-v` : Combines all the options together in one field, separated by `:` .
      - First field is the path to the file/directory on the **host machine**.
      - Second field is the path where the file/directory is mounted in the **container**.

    - Third field is optional, and is comma-separated list of options.
2. `--mount` : Consists of multiple key-value pairs, separated by commas each consisting of a `<key>=<value>` tuple. Some keys include:
    - `type` , which can be `bind` , `volume` , or `tmpfs` .
    - `source` , the path to the file/directory on the **host**.
    - `destination` , the path to the file/directory mounted in the **container**.

Docker Network

- Container networking refers to the ability for containers to connect to and communicate with each other, and with non-Docker network services.
- Users can create a docker network to connect containers to them, or connect containers to non-Docker workloads.
- Create a user-defined bridge network: `$ docker network create <name>`
- Connect a container to a network: `$ docker network connect <net-name> <container>`
- Use the `--network host` option when using `docker run` to share the host's network with container.
- To learn more: https://www.youtube.com/watch?v=zJD7QYQtiKc

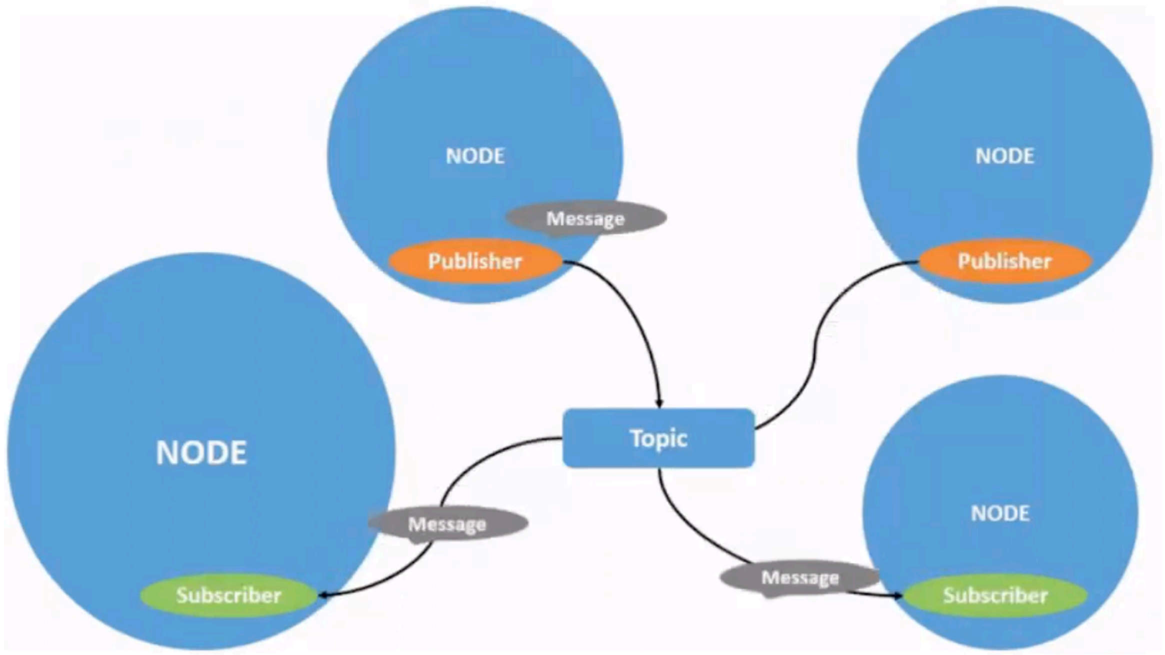
Docker Compose

- Compose is a tool for defining and running multiple containers at the same time.
- A yaml file `docker-compose.yml` is used to configure.
- Run `docker compose up` or `docker-compose up` (if you've installed the docker-compose) to start all your containers.

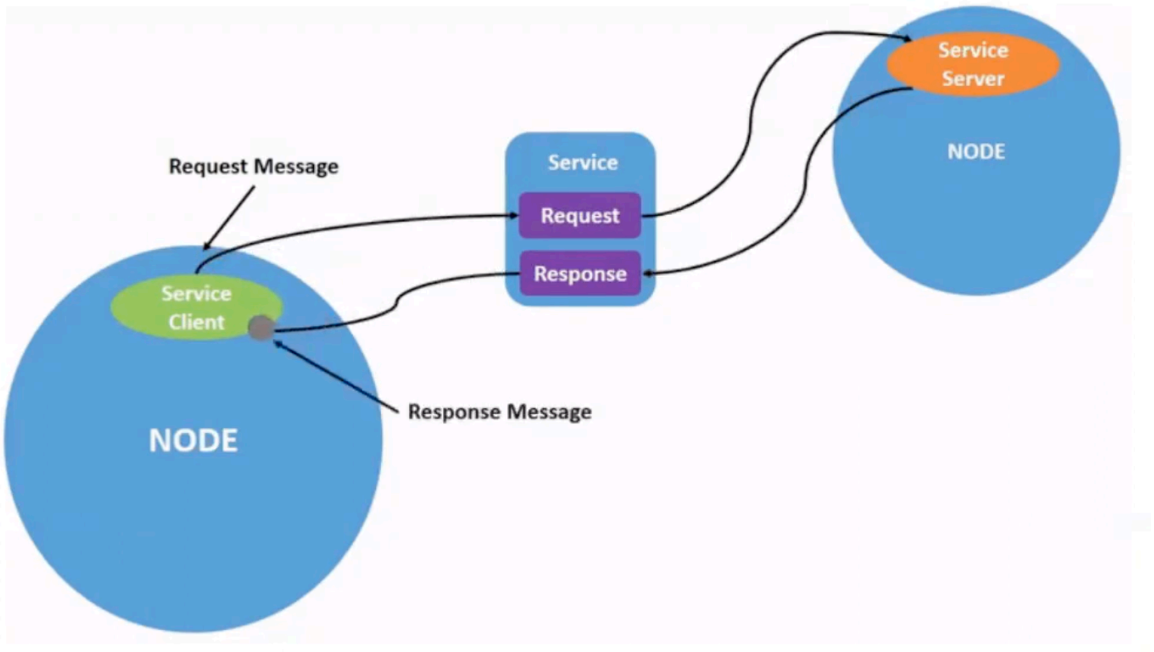
ROS 2

- **Robot Operating System (ROS) 2**:
    - **Distributed**: Utilizes a graph-like structure where nodes, which are individual software processes, communicate with each other through topics, services, and actions.
    - **Peer to peer**: Nodes communicate directly with each other
    - **Multi-lingual**: Supports C++ and Python
    - **Light-weight**
    - **Free and open-source**
- **ROS Graph**: The ROS graph is a network of ROS 2 elements processing data together at one time. It encompasses all executables and the connections between them if you were to map them all out and visualize them.
- **Nodes**:

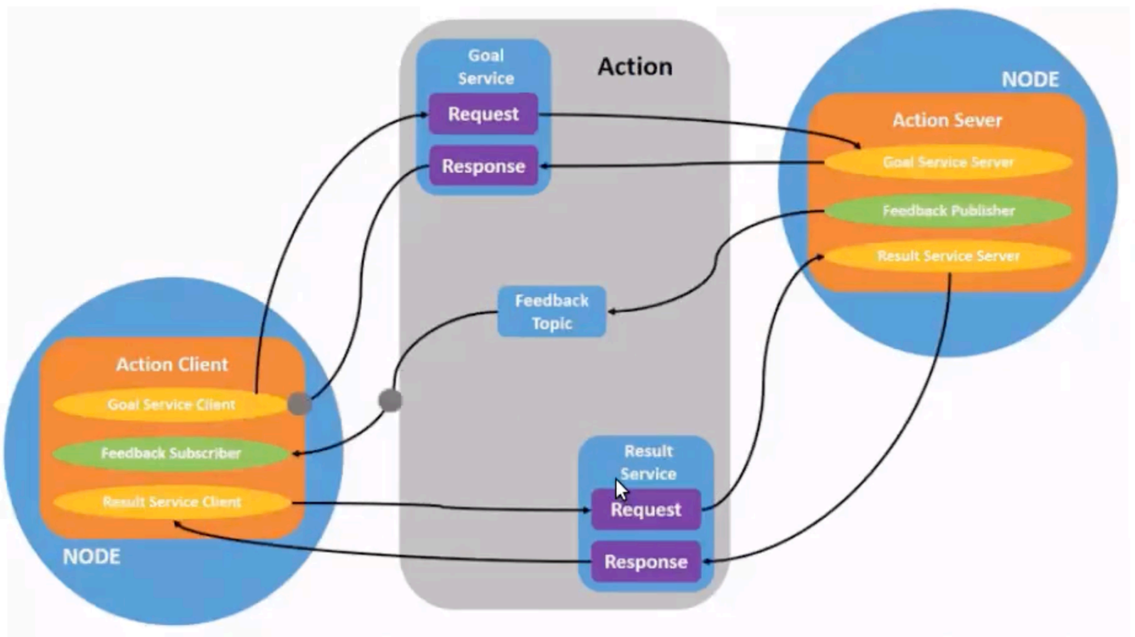
    - Each node in ROS 2 should be responsible for a single, module purpose (e.g., one node for controlling wheel motors, one node for controlling a laser range-finder, etc).
    - Each node can send and receive data to other nodes via topics, services, actions, or parameters.
    - To run a node: `ros 2 run <package_name> <exectuable_name>`
    - To list the nodes that are currently running: `ros2 node list`
    - To get information about a given node: `node info <node_name>`
- **Topics**: How nodes communicate with each other (main way data is moved between nodes). Acts as a bus for nodes to exchange messages.
    - A node **publishes** (sends) data to **topics**.
    - A node **subscribes** (receives) data from **topics**.
    - Topics don't have to only be point-to-point communication; it can be one-to-many, many-to-one, or many-to-many.
    - **Graph of how nodes and topics are communicating with each other, also shows the types of messages**: `rqt_graph`
    - **Lists all active topics**: `ros2 topic list`
    - **Lists all active topics with their message types**: `ros2 topic list -t`
    - **Displays messages being published on a topic**: `ros2 topic echo <topic_name>`
    - **Shows detailed information about a topic**: `ros2 topic info <topic_name>`
    - **Displays the structure of a message type**: `ros2 interface show <msg_type>`
    - **Publishes a message to a topic from the command line**: `ros2 topic pub <topic_name> <msg_type> '<args>'`
    - **Displays the publishing frequency of a topic**: `ros2 topic hz <topic_name>`
    - Diagram:



- **Services**: Services are another method of communication for nodes in the ROS graph. Services are based on call-and-response model, versus topics' publisher-subscriber model. While topics allow nodes to subscribe to data streams and get continual updates, **services only provide data when they are specifically called by a client**.
  - Can be many service clients using the same service. But **only one** service server for a service.
  - **Lists all active services**: `ros2 service list`
  - **Shows the type of a specific service**: `ros2 service type <service_name>`
  - **Lists all active services with their types**: `ros2 service list -t`
  - **Finds services by service type**: `ros2 service find <type_name>`
  - **Displays the structure of a service type (what type of data it needs for a request and what for response)**: `ros2 interface show <type_name>.srv`
  - **Calls a service from the command line**: `ros2 service call <service_name> <service_type> '<args>'`
  - Diagram:



- **Actions**: Actions are type of communication method meant for **long running tasks**. Consist of: a goal, feedback, and a result.
  - Actions are built on topics and services.
  - Unlike services, actions are preemptable (you can cancel them while executing) and provide steady feedback.
  - An "action client" node sends a goal to an "action server" node that acknowledges the goal and returns a stream of feedback and a result.
  - Diagram:



- **Parameters**: Configuration value for nodes (node settings).

- Each node maintains its own parameters (such as `int`, `float`, `bool`, etc).
- All parameters are dynamically reconfigurable and built off ROS 2 services.
- **Lists all parameters**: `ros2 param list`
- **Gets the value of a parameter**: `ros2 param get <node_name> <parameter_name>`
- **Sets the value of a parameter**: `ros2 param set <node_name> <parameter_name> <value>`
- Can also use YAML or launch file to define parameters: Useful to have all parameters in one place while tuning & set parameters for multiple nodes.
- Set in C++: `this->declare_parameter("my_parameter", "world");`
- Get in C++: `this->get_parameter("my_parameter");`
- Learn more here: https://roboticsbackend.com/rclcpp-params-tutorial-get-set-ros2-params-with-cpp/ & https://docs.ros.org/en/foxy/Tutorials/Beginner-Client-Libraries/Using-Parameters-In-A-Class-CPP.html

- **Workspace**: Directory containing all ROS 2 packages.
  - Before using ROS 2, you must source your ROS 2 installation workspace in the terminal you plan to work in to make them available for you to use in that terminal.
  - **Underlay**: Main ROS 2 environment.
  - Sourcing an **Overlay**: A secondary workspace where you can add new packages without interfering with the existing ROS 2 workspace that you are extending, or "underlay".
    - Your underlay must contain all the dependencies of all the packages in your overlay.
    - Packages in overlay override packages in underlay.
    - Possible to have several layers of underlays and overlays - where each successive overlay uses the packages of its parent underlays.
  - **Create a new workspace**: `$ mkdir -p <your_workspace>/src`. Then put desired ROS 2 packages inside src
  - **Resolving dependencies**: `$ cd <your_workspace>` `$ rosdep install -i --from-path src --rosdistro foxy -y`. This will install dependencies declared in `package.xml` from all packages in the `src` directory for ROS 2 foxy.
  - **Build workspace with `colcon`**: From the root of your workspace: `$ colcon build`.
    - `--packages-up-to`: Build packages you want, plus all its dependencies, but not the whole workspace. Saves time if you don't need all packages in workspace
  - Once build finishes, you should see: `build`, `install`, `log`, `src` directories in your workspace. The `install` directory is where your workspace's setup files are.
  - **Sourcing an Underlay**: Give your underlay access to your packages on top of the base ROS 2 environment.
    - Do it by writing in new terminal: `$ source /opt/ros/foxy/setup.bash`

        - Then, in root of desired workspace, `$ cd <your_workspace>` and `$ source isntall/local_setup.bash`
        - Instead of doing the above two notes you could combine them with: `$ install/setup.bash` in your workspace.
- **ROS 2 Packages**: A package can be considered a container in your ROS 2 code. To install and share your code, you'll need to organize it in a package.
    - **Package Creation**: ROS 2 uses **ament** as its build system and **colcon** as its build tool. Use CMake for your package.
    - CMake package requires:
        - File containing meta info about the package: `package.xml`
        - File describing how to build the code within the package: `CMakeLists.txt`
        - Most simple structure looks like folder named `my_package` with the above two files inside.
    - A single workspace can contain **many packages**, each in their own folder. You **cannot have nested packages**.
    - Best practice: Have src folder within your workspace and create all your packages in there.
    - **Create Package**: `$ cd <your_workspace>/src` and `$ ros 2 pkg create --build-type ament_cmake <package_name>`
    - **Package Contents**:
        - Contains files: `CMakeLists.txt`, and `package.xml`, and folders: `include`, `src`. Node source files (`.cpp`) are in src, and header files (`.hpp`) are in include.
    - **Customizing package.xml**: Fill in name and email on `maintainer` line, edit description to `summarize` the package, update the `license` line. Fill in your dependencies under the `_depend` tags. For more docs about what types of depend tags: https://docs.ros.org/en/humble/Tutorials/Intermediate/Rosdep.html
- Wow you made it all the way down here! If you need to learn more about anything covered above, you can always refer to the docs. Our club has a repository where we will continually add documentation and tutorials for you to use here: https://github.com/sfu-racerbot/racerbot-docs/tree/main