TurtleBot3 ROS2 Humble C++ Development Guide

Prerequisites

Before starting, ensure you have:

- ROS2 Humble installed
- TurtleBot3 packages installed
- A workspace set up (e.g., `~/turtlebot3_ws`)

```bash
# Create workspace if you don't have one
mkdir -p ~/turtlebot3_ws/src
cd ~/turtlebot3_ws
colcon build
source install/setup.bash
```

1. Creating a ROS2 Package

First, create a new package for your TurtleBot3 project:

```bash
cd ~/turtlebot3_ws/src
ros2 pkg create --build-type ament_cmake turtlebot3_custom --dependencies rclcpp std_msgs geometry_msgs sens
```

2. Creating Custom Interfaces

Custom Action Definition

Create `action/NavigateToGoal.action`:

```
# Goal
geometry_msgs/Pose target_pose
---
# Result
bool success
string message
geometry_msgs/Pose final_pose
---
# Feedback
geometry_msgs/Pose current_pose
float32 distance_remaining
float32 time_elapsed
```

Custom Service Definition

Create `srv/GetRobotStatus.srv`:

```
# Request
string query_type
---
# Response
bool success
string status_message
geometry_msgs/Pose current_pose
sensor_msgs/BatteryState battery_info
```

Update CMakeLists.txt for Custom Interfaces

Add to your `CMakeLists.txt`:

```cmake
find_package(rosidl_default_generators REQUIRED)

# Add custom interfaces
rosidl_generate_interfaces(${PROJECT_NAME}
  "action/NavigateToGoal.action"
  "srv/GetRobotStatus.srv"
  DEPENDENCIES geometry_msgs sensor_msgs
)
```

3. Creating a Basic Node

Basic TurtleBot3 Controller Node

Create `src/turtlebot3_controller.cpp`:

```cpp
```

```cpp
#include <rclcpp/rclcpp.hpp>
#include <geometry_msgs/msg/twist.hpp>
#include <sensor_msgs/msg/laser_scan.hpp>
#include <nav_msgs/msg/odometry.hpp>
#include <tf2/LinearMath/Quaternion.h>
#include <tf2_geometry_msgs/tf2_geometry_msgs.hpp>

class TurtleBot3Controller : public rclcpp::Node
{
public:
  TurtleBot3Controller() : Node("turtlebot3_controller")
  {
    // Publishers
    cmd_vel_pub_ = this->create_publisher<geometry_msgs::msg::Twist>("/cmd_vel", 10);

    // Subscribers
    laser_sub_ = this->create_subscription<sensor_msgs::msg::LaserScan>(
      "/scan", 10, std::bind(&TurtleBot3Controller::laser_callback, this, std::placeholders::_1));

    odom_sub_ = this->create_subscription<nav_msgs::msg::Odometry>(
      "/odom", 10, std::bind(&TurtleBot3Controller::odom_callback, this, std::placeholders::_1));

    // Timer for control loop
    control_timer_ = this->create_wall_timer(
      std::chrono::milliseconds(100), std::bind(&TurtleBot3Controller::control_loop, this));

    RCLCPP_INFO(this->get_logger(), "TurtleBot3 Controller Node Started");
  }

private:
  void laser_callback(const sensor_msgs::msg::LaserScan::SharedPtr msg)
  {
    // Find minimum distance in front of robot
    size_t front_index = msg->ranges.size() / 2;
    size_t range = 30; // Check 30 degrees in front

    front_distance_ = std::numeric_limits<float>::max();
    for (size_t i = front_index - range; i <= front_index + range; ++i) {
      if (i < msg->ranges.size() && msg->ranges[i] > 0.0) {
        front_distance_ = std::min(front_distance_, msg->ranges[i]);
      }
    }
  }
```

```
  void odom_callback(const nav_msgs::msg::Odometry::SharedPtr msg)
  {
    current_pose_ = msg->pose.pose;

    // Extract yaw from quaternion
    tf2::Quaternion quat;
    tf2::fromMsg(current_pose_.orientation, quat);
    double roll, pitch, yaw;
    tf2::Matrix3x3(quat).getRPY(roll, pitch, yaw);
    current_yaw_ = yaw;
  }

  void control_loop()
  {
    auto twist = geometry_msgs::msg::Twist();

    // Simple obstacle avoidance
    if (front_distance_ > 0.5) {
      twist.linear.x = 0.2;  // Move forward
      twist.angular.z = 0.0;
    } else {
      twist.linear.x = 0.0;
      twist.angular.z = 0.5; // Turn right
    }

    cmd_vel_pub_->publish(twist);
  }

  // Member variables
  rclcpp::Publisher<geometry_msgs::msg::Twist>::SharedPtr cmd_vel_pub_;
  rclcpp::Subscription<sensor_msgs::msg::LaserScan>::SharedPtr laser_sub_;
  rclcpp::Subscription<nav_msgs::msg::Odometry>::SharedPtr odom_sub_;
  rclcpp::TimerBase::SharedPtr control_timer_;

  float front_distance_ = std::numeric_limits<float>::max();
  geometry_msgs::msg::Pose current_pose_;
  double current_yaw_ = 0.0;
};

int main(int argc, char** argv)
{
  rclcpp::init(argc, argv);
  rclcpp::spin(std::make_shared<TurtleBot3Controller>());
```

```
    rclcpp::shutdown();
    return 0;
}
```

4. Creating a Service Server

Service Server Node

Create `src/robot_status_server.cpp`:

cpp

```cpp
#include <rclcpp/rclcpp.hpp>
#include <geometry_msgs/msg/pose.hpp>
#include <sensor_msgs/msg/battery_state.hpp>
#include <nav_msgs/msg/odometry.hpp>
#include "turtlebot3_custom/srv/get_robot_status.hpp"

class RobotStatusServer : public rclcpp::Node
{
public:
  RobotStatusServer() : Node("robot_status_server")
  {
    // Service server
    status_service_ = this->create_service<turtlebot3_custom::srv::GetRobotStatus>(
      "get_robot_status",
      std::bind(&RobotStatusServer::handle_status_request, this,
          std::placeholders::_1, std::placeholders::_2));

    // Subscribers to get robot state
    odom_sub_ = this->create_subscription<nav_msgs::msg::Odometry>(
      "/odom", 10, std::bind(&RobotStatusServer::odom_callback, this, std::placeholders::_1));

    battery_sub_ = this->create_subscription<sensor_msgs::msg::BatteryState>(
      "/battery_state", 10, std::bind(&RobotStatusServer::battery_callback, this, std::placeholders::_1));

    RCLCPP_INFO(this->get_logger(), "Robot Status Server ready");
  }

private:
  void handle_status_request(const std::shared_ptr<turtlebot3_custom::srv::GetRobotStatus::Request> request,
                std::shared_ptr<turtlebot3_custom::srv::GetRobotStatus::Response> response)
  {
    RCLCPP_INFO(this->get_logger(), "Received status request: %s", request->query_type.c_str());

    response->success = true;
    response->current_pose = current_pose_;
    response->battery_info = battery_state_;

    if (request->query_type == "position") {
      response->status_message = "Current position: x=" +
        std::to_string(current_pose_.position.x) +
        ", y=" + std::to_string(current_pose_.position.y);
    } else if (request->query_type == "battery") {
      response->status_message = "Battery level: " +
```

```
            std::to_string(battery_state_.percentage * 100) + "%";
        } else {
            response->status_message = "Robot is operational. Position and battery data available.";
        }
    }

    void odom_callback(const nav_msgs::msg::Odometry::SharedPtr msg)
    {
        current_pose_ = msg->pose.pose;
    }

    void battery_callback(const sensor_msgs::msg::BatteryState::SharedPtr msg)
    {
        battery_state_ = *msg;
    }

    // Member variables
    rclcpp::Service<turtlebot3_custom::srv::GetRobotStatus>::SharedPtr status_service_;
    rclcpp::Subscription<nav_msgs::msg::Odometry>::SharedPtr odom_sub_;
    rclcpp::Subscription<sensor_msgs::msg::BatteryState>::SharedPtr battery_sub_;

    geometry_msgs::msg::Pose current_pose_;
    sensor_msgs::msg::BatteryState battery_state_;
};

int main(int argc, char** argv)
{
    rclcpp::init(argc, argv);
    rclcpp::spin(std::make_shared<RobotStatusServer>());
    rclcpp::shutdown();
    return 0;
}
```

Service Client Example

Create `src/status_client.cpp`:

```
cpp
```

```cpp
#include <rclcpp/rclcpp.hpp>
#include "turtlebot3_custom/srv/get_robot_status.hpp"

class StatusClient : public rclcpp::Node
{
public:
  StatusClient() : Node("status_client")
  {
    client_ = this->create_client<turtlebot3_custom::srv::GetRobotStatus>("get_robot_status");

    // Wait for service to be available
    while (!client_->wait_for_service(std::chrono::seconds(1))) {
      if (!rclcpp::ok()) {
        RCLCPP_ERROR(rclcpp::get_logger("rclcpp"), "Interrupted while waiting for service");
        return;
      }
      RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "Service not available, waiting...");
    }

    send_request();
  }

private:
  void send_request()
  {
    auto request = std::make_shared<turtlebot3_custom::srv::GetRobotStatus::Request>();
    request->query_type = "general";

    auto result = client_->async_send_request(request);

    // Wait for the result
    if (rclcpp::spin_until_future_complete(this->get_node_base_interface(), result) ==
      rclcpp::FutureReturnCode::SUCCESS)
    {
      auto response = result.get();
      RCLCPP_INFO(this->get_logger(), "Response: %s", response->status_message.c_str());
      RCLCPP_INFO(this->get_logger(), "Success: %s", response->success ? "true" : "false");
    } else {
      RCLCPP_ERROR(this->get_logger(), "Failed to call service");
    }
  }

  rclcpp::Client<turtlebot3_custom::srv::GetRobotStatus>::SharedPtr client_;
```

```
};

int main(int argc, char** argv)
{
  rclcpp::init(argc, argv);
  rclcpp::spin(std::make_shared<StatusClient>());
  rclcpp::shutdown();
  return 0;
}
```

5. Creating an Action Server

Action Server Node

Create `src/navigation_action_server.cpp`:

```
cpp
```

```cpp
#include <functional>
#include <memory>
#include <thread>

#include <rclcpp/rclcpp.hpp>
#include <rclcpp_action/rclcpp_action.hpp>
#include <geometry_msgs/msg/twist.hpp>
#include <nav_msgs/msg/odometry.hpp>
#include <tf2/LinearMath/Quaternion.h>
#include <tf2_geometry_msgs/tf2_geometry_msgs.hpp>

#include "turtlebot3_custom/action/navigate_to_goal.hpp"

class NavigationActionServer : public rclcpp::Node
{
public:
  using NavigateToGoal = turtlebot3_custom::action::NavigateToGoal;
  using GoalHandleNavigation = rclcpp_action::ServerGoalHandle<NavigateToGoal>;

  NavigationActionServer() : Node("navigation_action_server")
  {
    // Action server
    action_server_ = rclcpp_action::create_server<NavigateToGoal>(
      this,
      "navigate_to_goal",
      std::bind(&NavigationActionServer::handle_goal, this, std::placeholders::_1, std::placeholders::_2),
      std::bind(&NavigationActionServer::handle_cancel, this, std::placeholders::_1),
      std::bind(&NavigationActionServer::handle_accepted, this, std::placeholders::_1));

    // Publishers and subscribers
    cmd_vel_pub_ = this->create_publisher<geometry_msgs::msg::Twist>("/cmd_vel", 10);
    odom_sub_ = this->create_subscription<nav_msgs::msg::Odometry>(
      "/odom", 10, std::bind(&NavigationActionServer::odom_callback, this, std::placeholders::_1));

    RCLCPP_INFO(this->get_logger(), "Navigation Action Server ready");
  }

private:
  rclcpp_action::GoalResponse handle_goal(
    const rclcpp_action::GoalUUID & uuid,
    std::shared_ptr<const NavigateToGoal::Goal> goal)
  {
    RCLCPP_INFO(this->get_logger(), "Received goal request");
```

```
    (void)uuid;
    return rclcpp_action::GoalResponse::ACCEPT_AND_EXECUTE;
  }

  rclcpp_action::CancelResponse handle_cancel(
    const std::shared_ptr<GoalHandleNavigation> goal_handle)
  {
    RCLCPP_INFO(this->get_logger(), "Received request to cancel goal");
    (void)goal_handle;
    return rclcpp_action::CancelResponse::ACCEPT;
  }

  void handle_accepted(const std::shared_ptr<GoalHandleNavigation> goal_handle)
  {
    // Execute the goal in a separate thread
    std::thread{std::bind(&NavigationActionServer::execute, this, std::placeholders::_1), goal_handle}.detach();
  }

  void execute(const std::shared_ptr<GoalHandleNavigation> goal_handle)
  {
    RCLCPP_INFO(this->get_logger(), "Executing goal");
    rclcpp::Rate loop_rate(10);
    const auto goal = goal_handle->get_goal();
    auto feedback = std::make_shared<NavigateToGoal::Feedback>();
    auto result = std::make_shared<NavigateToGoal::Result>();

    auto start_time = this->now();

    // Simple navigation towards goal
    while (rclcpp::ok()) {
      // Check if goal is canceled
      if (goal_handle->is_canceling()) {
        result->success = false;
        result->message = "Goal was canceled";
        result->final_pose = current_pose_;
        goal_handle->canceled(result);
        RCLCPP_INFO(this->get_logger(), "Goal canceled");
        return;
      }

      // Calculate distance to goal
      double dx = goal->target_pose.position.x - current_pose_.position.x;
      double dy = goal->target_pose.position.y - current_pose_.position.y;
      double distance = sqrt(dx*dx + dy*dy);
```

```
      // Update feedback
      feedback->current_pose = current_pose_;
      feedback->distance_remaining = distance;
      feedback->time_elapsed = (this->now() - start_time).seconds();
      goal_handle->publish_feedback(feedback);

      // Check if goal is reached
      if (distance < 0.1) {
        result->success = true;
        result->message = "Goal reached successfully";
        result->final_pose = current_pose_;
        goal_handle->succeed(result);
        RCLCPP_INFO(this->get_logger(), "Goal succeeded");
        return;
      }

      // Simple proportional controller
      auto twist = geometry_msgs::msg::Twist();
      twist.linear.x = std::min(0.2, distance * 0.5);
      twist.angular.z = atan2(dy, dx) - current_yaw_;

      // Normalize angular velocity
      while (twist.angular.z > M_PI) twist.angular.z -= 2*M_PI;
      while (twist.angular.z < -M_PI) twist.angular.z += 2*M_PI;
      twist.angular.z *= 0.5;

      cmd_vel_pub_->publish(twist);
      loop_rate.sleep();
    }
}

void odom_callback(const nav_msgs::msg::Odometry::SharedPtr msg)
{
    current_pose_ = msg->pose.pose;

    // Extract yaw from quaternion
    tf2::Quaternion quat;
    tf2::fromMsg(current_pose_.orientation, quat);
    double roll, pitch, yaw;
    tf2::Matrix3x3(quat).getRPY(roll, pitch, yaw);
    current_yaw_ = yaw;
}
```

```
  rclcpp_action::Server<NavigateToGoal>::SharedPtr action_server_;
  rclcpp::Publisher<geometry_msgs::msg::Twist>::SharedPtr cmd_vel_pub_;
  rclcpp::Subscription<nav_msgs::msg::Odometry>::SharedPtr odom_sub_;

  geometry_msgs::msg::Pose current_pose_;
  double current_yaw_ = 0.0;
};

int main(int argc, char ** argv)
{
  rclcpp::init(argc, argv);
  rclcpp::spin(std::make_shared<NavigationActionServer>());
  rclcpp::shutdown();
  return 0;
}
```

Action Client Example

Create `src/navigation_client.cpp`:

```
cpp
```

```cpp
#include <functional>
#include <future>
#include <memory>
#include <string>
#include <sstream>

#include <rclcpp/rclcpp.hpp>
#include <rclcpp_action/rclcpp_action.hpp>

#include "turtlebot3_custom/action/navigate_to_goal.hpp"

class NavigationActionClient : public rclcpp::Node
{
public:
  using NavigateToGoal = turtlebot3_custom::action::NavigateToGoal;
  using GoalHandleNavigation = rclcpp_action::ClientGoalHandle<NavigateToGoal>;

  NavigationActionClient() : Node("navigation_action_client")
  {
    client_ = rclcpp_action::create_client<NavigateToGoal>(
      this,
      "navigate_to_goal");

    send_goal();
  }

  void send_goal()
  {
    if (!client_->wait_for_action_server()) {
      RCLCPP_ERROR(this->get_logger(), "Action server not available after waiting");
      rclcpp::shutdown();
    }

    auto goal_msg = NavigateToGoal::Goal();
    goal_msg.target_pose.position.x = 2.0;
    goal_msg.target_pose.position.y = 1.0;
    goal_msg.target_pose.position.z = 0.0;

    RCLCPP_INFO(this->get_logger(), "Sending goal");

    auto send_goal_options = rclcpp_action::Client<NavigateToGoal>::SendGoalOptions();
    send_goal_options.goal_response_callback =
      std::bind(&NavigationActionClient::goal_response_callback, this, std::placeholders::_1);
```

```
    send_goal_options.feedback_callback =
      std::bind(&NavigationActionClient::feedback_callback, this, std::placeholders::_1, std::placeholders::_2);
    send_goal_options.result_callback =
      std::bind(&NavigationActionClient::result_callback, this, std::placeholders::_1);
    client_->async_send_goal(goal_msg, send_goal_options);
  }

private:
  rclcpp_action::Client<NavigateToGoal>::SharedPtr client_;

  void goal_response_callback(const GoalHandleNavigation::SharedPtr & goal_handle)
  {
    if (!goal_handle) {
      RCLCPP_ERROR(this->get_logger(), "Goal was rejected by server");
    } else {
      RCLCPP_INFO(this->get_logger(), "Goal accepted by server, waiting for result");
    }
  }

  void feedback_callback(
    GoalHandleNavigation::SharedPtr,
    const std::shared_ptr<const NavigateToGoal::Feedback> feedback)
  {
    RCLCPP_INFO(this->get_logger(), "Distance remaining: %.2f", feedback->distance_remaining);
  }

  void result_callback(const GoalHandleNavigation::WrappedResult & result)
  {
    switch (result.code) {
      case rclcpp_action::ResultCode::SUCCEEDED:
        RCLCPP_INFO(this->get_logger(), "Goal succeeded: %s", result.result->message.c_str());
        break;
      case rclcpp_action::ResultCode::ABORTED:
        RCLCPP_ERROR(this->get_logger(), "Goal was aborted");
        break;
      case rclcpp_action::ResultCode::CANCELED:
        RCLCPP_ERROR(this->get_logger(), "Goal was canceled");
        break;
      default:
        RCLCPP_ERROR(this->get_logger(), "Unknown result code");
        break;
    }
    rclcpp::shutdown();
  }
```

```
};

int main(int argc, char ** argv)
{
  rclcpp::init(argc, argv);
  rclcpp::spin(std::make_shared<NavigationActionClient>());
  rclcpp::shutdown();
  return 0;
}
```

6. Launch Files

Main Launch File

Create `launch/turtlebot3_custom.launch.py`:

```
python
```

```python
from launch import LaunchDescription
from launch_ros.actions import Node
from launch.actions import DeclareLaunchArgument, ExecuteProcess
from launch.substitutions import LaunchConfiguration
from launch.conditions import IfCondition

def generate_launch_description():
    # Declare launch arguments
    use_sim_time = LaunchConfiguration('use_sim_time', default='false')
    robot_model = LaunchConfiguration('robot_model', default='burger')

    return LaunchDescription([
        # Launch arguments
        DeclareLaunchArgument(
            'use_sim_time',
            default_value='false',
            description='Use simulation time if true'
        ),

        DeclareLaunchArgument(
            'robot_model',
            default_value='burger',
            description='TurtleBot3 model (burger, waffle, waffle_pi)'
        ),

        DeclareLaunchArgument(
            'start_rviz',
            default_value='false',
            description='Start RViz'
        ),

        # TurtleBot3 Controller Node
        Node(
            package='turtlebot3_custom',
            executable='turtlebot3_controller',
            name='turtlebot3_controller',
            parameters=[{'use_sim_time': use_sim_time}],
            output='screen'
        ),

        # Robot Status Server
        Node(
            package='turtlebot3_custom',
```

```
        executable='robot_status_server',
        name='robot_status_server',
        parameters=[{'use_sim_time': use_sim_time}],
        output='screen'
    ),

    # Navigation Action Server
    Node(
        package='turtlebot3_custom',
        executable='navigation_action_server',
        name='navigation_action_server',
        parameters=[{'use_sim_time': use_sim_time}],
        output='screen'
    ),

    # Optional: Start RViz
    Node(
        package='rviz2',
        executable='rviz2',
        name='rviz2',
        condition=IfCondition(LaunchConfiguration('start_rviz')),
        output='screen'
    ),
])
```

Simulation Launch File

Create `launch/turtlebot3_simulation.launch.py`:

```
python
```

```python
import os
from launch import LaunchDescription
from launch.actions import IncludeLaunchDescription, DeclareLaunchArgument
from launch.launch_description_sources import PythonLaunchDescriptionSource
from launch.substitutions import LaunchConfiguration
from launch_ros.substitutions import FindPackageShare

def generate_launch_description():
    # Get the launch directory
    pkg_gazebo_ros = FindPackageShare(package='gazebo_ros').find('gazebo_ros')
    pkg_turtlebot3_gazebo = FindPackageShare(package='turtlebot3_gazebo').find('turtlebot3_gazebo')

    # Launch configuration variables
    use_sim_time = LaunchConfiguration('use_sim_time', default='true')
    world = LaunchConfiguration('world')

    # Declare the launch arguments
    declare_world_cmd = DeclareLaunchArgument(
      'world',
      default_value=os.path.join(pkg_turtlebot3_gazebo, 'worlds', 'turtlebot3_world.world'),
      description='Full path to world model file to load')

    # Start Gazebo server
    start_gazebo_server_cmd = IncludeLaunchDescription(
      PythonLaunchDescriptionSource(os.path.join(pkg_gazebo_ros, 'launch', 'gzserver.launch.py')),
      launch_arguments={'world': world}.items())

    # Start Gazebo client
    start_gazebo_client_cmd = IncludeLaunchDescription(
      PythonLaunchDescriptionSource(os.path.join(pkg_gazebo_ros, 'launch', 'gzclient.launch.py')))

    # Robot State Publisher
    robot_state_publisher_cmd = IncludeLaunchDescription(
      PythonLaunchDescriptionSource(os.path.join(pkg_turtlebot3_gazebo, 'launch', 'robot_state_publisher.launch.py
      launch_arguments={'use_sim_time': use_sim_time}.items())

    # Spawn TurtleBot3
    spawn_turtlebot_cmd = IncludeLaunchDescription(
      PythonLaunchDescriptionSource(os.path.join(pkg_turtlebot3_gazebo, 'launch', 'spawn_turtlebot3.launch.py')),
      launch_arguments={'use_sim_time': use_sim_time}.items())

    # Include custom nodes
    custom_nodes_cmd = IncludeLaunchDescription(
```

```
    PythonLaunchDescriptionSource(os.path.join(FindPackageShare('turtlebot3_custom').find('turtlebot3_custom')
    launch_arguments={'use_sim_time': use_sim_time}.items())

# Create the launch description and populate
ld = LaunchDescription()

# Add the commands to the launch description
ld.add_action(declare_world_cmd)
ld.add_action(start_gazebo_server_cmd)
ld.add_action(start_gazebo_client_cmd)
ld.add_action(robot_state_publisher_cmd)
ld.add_action(spawn_turtlebot_cmd)
ld.add_action(custom_nodes_cmd)

return ld
```

7. Complete CMakeLists.txt

Update your `CMakeLists.txt`:

```
cmake
```

```
cmake_minimum_required(VERSION 3.8)
project(turtlebot3_custom)

if(CMAKE_COMPILER_IS_GNUCXX OR CMAKE_CXX_COMPILER_ID MATCHES "Clang")
  add_compile_options(-Wall -Wextra -Wpedantic)
endif()

# Find dependencies
find_package(ament_cmake REQUIRED)
find_package(rclcpp REQUIRED)
find_package(rclcpp_action REQUIRED)
find_package(std_msgs REQUIRED)
find_package(geometry_msgs REQUIRED)
find_package(sensor_msgs REQUIRED)
find_package(nav_msgs REQUIRED)
find_package(tf2 REQUIRED)
find_package(tf2_ros REQUIRED)
find_package(tf2_geometry_msgs REQUIRED)
find_package(rosidl_default_generators REQUIRED)

# Generate interfaces
rosidl_generate_interfaces(${PROJECT_NAME}
  "action/NavigateToGoal.action"
  "srv/GetRobotStatus.srv"
  DEPENDENCIES geometry_msgs sensor_msgs
)

# Add executables
add_executable(turtlebot3_controller src/turtlebot3_controller.cpp)
add_executable(robot_status_server src/robot_status_server.cpp)
add_executable(status_client src/status_client.cpp)
add_executable(navigation_action_server src/navigation_action_server.cpp)
add_executable(navigation_client src/navigation_client.cpp)

# Dependencies for executables
ament_target_dependencies(turtlebot3_controller
  rclcpp std_msgs geometry_msgs sensor_msgs nav_msgs tf2 tf2_ros tf2_geometry_msgs)

ament_target_dependencies(robot_status_server
  rclcpp geometry_msgs sensor_msgs nav_msgs)

ament_target_dependencies(status_client
  rclcpp)
```

```
ament_target_dependencies(navigation_action_server
 rclcpp rclcpp_action geometry_msgs nav_msgs tf2 tf2_geometry_msgs)

ament_target_dependencies(navigation_client
 rclcpp rclcpp_action)

# Link custom interfaces
rosidl_target_interfaces(robot_status_server ${PROJECT_NAME} "rosidl_typesupport_cpp")
rosidl_target_interfaces(status_client ${PROJECT_NAME} "rosidl_typesupport_cpp")
rosidl_target_interfaces(navigation_action_server ${PROJECT_NAME} "rosidl_typesupport_cpp")
rosidl_target_interfaces(navigation_client ${
```