

# **Developing a ROS 2 Action package**

**Kheng Lee Koay**

**February 17, 2026**

# Contents

<b>1</b>	<b>Developing a ROS 2 Actions</b>	<b>3</b>
1.1	ROS 2 Publisher/Subscribers, Services and Actions . . . . .	3
1.2	Introduction to ROS 2 Actions . . . . .	3
1.3	How ROS 2 Actions Work . . . . .	4
1.4	ROS 2 Action Interface Definition . . . . .	5
1.5	Building and Testing a ROS Action Interface, Server, and Client . .	6
1.5.1	Building a custom action interface . . . . .	6
1.5.2	Create the Action Package with an Action Server and Client .	8
1.5.3	Test the Action Package . . . . .	12
1.6	Homework: Implementing Goal Cancellation . . . . .	13

# Chapter 1 Developing a ROS 2 Actions

## 1.1 ROS 2 Publisher/Subscribers, Services and Actions

In the ROS 2 framework, there are multiple ways to facilitate communication between different components of a robotic system. So far, you have explored two primary communication methods:

### 1. Publishers and Subscribers (Topics):

- ✿ Used for continuous, unidirectional data flow.
- ✿ A publisher sends data, and a subscriber receives and processes it.
- ✿ Ideal for streaming sensor data or status updates.

### 2. Services:

- ✿ Used for synchronous, request-response communication.
- ✿ A client sends a request to a server, and the server returns a response.
- ✿ Useful for tasks requiring immediate actions, such as turning on an LED.

While both methods are effective, they have limitations in handling long-running tasks and real-time feedback. For instance, if a mobile robot is navigating to a target location, it needs:

- ✿ The ability to cancel the task if required.
- ✿ Real-time feedback on its current position.
- ✿ Concurrent handling of multiple action requests.

To address these requirements, ROS 2 Actions provide an enhanced communication mechanism.

## 1.2 Introduction to ROS 2 Actions

ROS 2 Actions are designed for asynchronous, long-running tasks. An action consists of three services and two topics (shown below) that allow communication between an **Action Client** and an **Action Server**.

### Services:

- `send_goal` - Initiates a task (e.g. `goto "target_distance: 10.0"`).
- `cancel_goal` - Cancels an active task.
- `get_result` - Requests the outcome of a completed task.

## Topics:

`feedback` - Provides real-time updates during task execution.

`status` - Shares the status of ongoing tasks.

The diagram in figure 1.1 illustrates the communication channels between an Action Client and an Action Server along with the corresponding message types for each channel. For detailed explanation see [Actions](#).



Figure 1.1. An overview of the communication channels and message types between an Action Client and an Action Server. Note: Bolded message types represent user-defined message types specified in the Action Interface Definition (.action) file.

## 1.3 How ROS 2 Actions Work

Figure 1.2 below illustrates a typical ROS 2 Action Client/Server interaction. The process begins when the **Action Client** sends a `goal request` to the **Action Server**. This goal request uses a user-defined message type specified in the `.action` definition file.

The **Action Server** evaluates the request and responds synchronously with a goal response message, in this example the goal is accepted and the time it was accepted. Once the goal is accepted, the user-defined execution method (which contains the algorithm or logic for performing the task) is triggered on the **Action Server**.

Immediately after receiving the `goal acceptance` message, the **Action Client** makes an asynchronous `result request` to retrieve the result when the task is completed.

As the task is being executed, the **Action Server** continuously publishes `feedback` messages, these messages are also based on a user-defined message type and are generated by a user-defined feedback method. This allows the **Action Client** to monitor real-time progress. When the task is completed, the **Action Server** sends the `result` message to the **Action Client**.

Both the content of the result and the method to produce it are user-defined, allowing complete customisation of the action's behaviour and output.

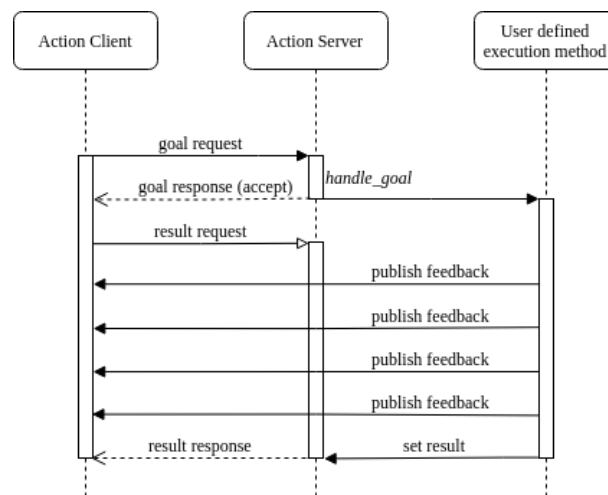


Figure 1.2. An example of interaction between ROS 2 Action Server and Action Client (image from <https://design.ros2.org/articles/actions.html>)

## 1.4 ROS 2 Action Interface Definition

ROS 2 Actions use a dedicated file format (`.action`) to define the structure of the communication between the **Action Client** and **Action Server**. This file is divided into three sections, each separated by `---`. The first section specifies the `Goal` (the input sent from the client to the server to initiate a task), the second defines the `Result` (the output returned upon task completion), and the third outlines the `Feedback` (real-time progress updates sent during task execution). Below is an example `MoveTo.action` file demonstrating how these message types are defined.

```

# Goal definition
float32 target_distance
---
# Result definition
float32 total_distance_travelled
---
```

```
# Feedback definition
float32 current_distance_travelled
```

---

## 1.5 Building and Testing a ROS Action Interface, Server, and Client

In this section, you will create a ROS 2 Action communication protocol to simulate the interaction between a mobile robot (Action Server) and its control system (Action Client). The control system will send requests for the robot to move a specified distance forward. During movement, the robot will provide real-time progress updates and return the final result once the task is complete. You will:

- ❁ Add a custom action interface to existing `my_custom_interfaces` package.
- ❁ Create a new ROS 2 package called `robot_actions` for the **Action Server** and **Action Client** using the `ament_python` build type.
- ❁ Implement and test the **Action Server** and **Action Client**.

This practical builds on your previous ROS 2 knowledge, including:

- ❁ ROS 2 Workspace and Package Creation (see DOC03)
- ❁ Creating custom ROS 2 interfaces (see DOC04 Appendix 1)

Additionally, you will learn the following new skills:

- ❁ How to add an **Action Interface** (`.action` file) to a package (`my_custom_interfaces`)
- ❁ How to implementing both an **Action Server** and an **Action Client**.
- ❁ How to test the interaction between the **Action Client** and **Action Server**.

### 1.5.1 Building a custom action interface

#### 1. Create an Action Interface

You will create an Action Definition File and store it in the action directory of your `my_custom_interfaces` package (created as part of Chapter ?? exercise). The advantage of defining the Action Interface in a separate package from the Action Server is that it allows the Action Client to integrate it as an independent dependency without requiring the installation of all the dependencies needed for the Action Server to function.

- Navigate to your `my_custom_interfaces` package:

```
~$ cd Workspaces/ros2_ws/src/my_custom_interfaces
```

**b. Create an action directory and the MoveTo.action file in the directory**

```
~/Workspaces/ros2_ws/src/my_custom_interfaces$ mkdir action
~/Workspaces/ros2_ws/src/my_custom_interfaces$ cd action
~/Workspaces/ros2_ws/src/my_custom_interfaces/action$ touch \
    MoveTo.action
```

**c. Edit MoveTo.action file and add:**

```
# Goal definition
float32 target_distance
---
# Result definition
float32 total_distance_travelled
---
# Feedback definition
float32 current_distance_travelled
```

Listing 1.1. Content of MoveTo.action file.

**d. Update CMakeLists.txt in my\_custom\_interfaces to build language-specific code for the action/MoveTo.action interface**

```
find_package(rosidl_default_generators REQUIRED)

rosidl_generate_interfaces(${PROJECT_NAME}
  "srv/AddThreeInts.srv"
  "action/MoveTo.action"
)
```

Listing 1.2. Enable code generation for MoveTo.action Action Interfaces in CMakeLists.txt

**e. The action\_msgs package defines standard interfaces (supporting message types) used to support action communication in ROS 2. Therefore we need to update the package.xml to add action\_msgs dependency:**

```
<exec_depend>rosidl_default_runtime</exec_depend>
<depend>action_msgs</depend>
<member_of_group>rosidl_interface_packages</member_of_group>
```

Listing 1.3. Add action\_msgs package dependency to package.xml.

**f. Build the package:**

```
~$ cd Workspaces/ros2_ws/  
~/Workspaces/ros2_ws$ colcon build
```

- g. Verify the action interface was successfully created:

```
~/Workspaces/ros2_ws$ source install/setup.bash  
~/Workspaces/ros2_ws$ ros2 interface show \  
    my_custom_interfaces/action/MoveTo
```

## 1.5.2 Create the Action Package with an Action Server and Client

### 1. Create the `robot_action` Package

Next, create the `robot_actions` package, which will contain both your **Action Server** and **Action Client**. This package will be licensed under **Apache-2.0** and will depend on the `my_custom_interfaces` package's action interface (`MoveTo.action`) you previously built for the **Action Server**.

- a. Create a new package for the **Action Server** and **Client**:

```
~$ cd Workspaces/ros2_ws/src  
~/Workspaces/ros2_ws/src$ ros2 pkg create --build-type \  
    ament_python --license Apache-2.0 robot_actions --dependencies \  
    rclpy my_custom_interfaces
```

### 2. Implement the Action Server

- a. Inside your `robot_actions` package, create a new file named `robot_action_server.py` within the `robot_actions` subdirectory (typically where your node scripts are located).

```
~/Workspaces/ros2_ws/src/robot_actions/robot_actions$ touch \  
    robot_action_server.py
```

- b. Open the file in your code editor and implement the Action Server code as shown in figure below. Take the time to thoroughly understand the code provided. To improve your comprehension, add comments explaining key sections. If you need assistance, refer to the official ROS 2 documentation for guidance.

```
import rclpy  
from rclpy.action import ActionServer, CancelResponse, GoalResponse  
from rclpy.node import Node
```

```

from my_custom_interfaces.action import MoveTo
class RobotActionServer(Node):
    def __init__(self):
        super().__init__('robot_action_server')
        self._action_server = ActionServer(self, MoveTo, 'move_to', self.
            ↳ execute_callback)

    def execute_callback(self, goal_handle):
        self.get_logger().info('Executing goal...')
        feedback_msg = MoveTo.Feedback()
        feedback_msg.current_distance_travelled = 0.0

        while feedback_msg.current_distance_travelled < goal_handle.
            ↳ request.target_distance:
                feedback_msg.current_distance_travelled +=1.0
                self.get_logger().info(f'Feedback:{feedback_msg.
                    ↳ current_distance_travelled}m')
                goal_handle.publish_feedback(feedback_msg)
                rclpy.spin_once(self, timeout_sec=1)

        goal_handle.succeed()

        result = MoveTo.Result()
        result.total_distance_travelled = feedback_msg.
            ↳ current_distance_travelled
        return result

def main(args=None):
    rclpy.init(args=args)
    robot_action_server = RobotActionServer()
    print("Action Server Running...")
    try:
        while rclpy.ok():
            rclpy.spin_once(robot_action_server)
    except KeyboardInterrupt:
        robot_action_server._action_server.destroy()
        robot_action_server.destroy_node()

```

```
if __name__ == '__main__':  
    main()
```

---

Listing 1.4. Code listing for robot\_action\_server.py.

### 3. Implement the Action Client

a. Create robot\_action\_client.py

```
~/Workspaces/ros2_ws/src/robot_actions/robot_actions$ touch \  
robot_action_client.py
```

b. Implement the Action Client script as shown in figure below. Carefully review the provided code to ensure a thorough understanding, and enhance your comprehension by adding comments to explain key sections. If you need assistance, refer to the official ROS 2 documentation on [Writing an Action Server-Client](#) for guidance.

```
#robot_action_client.py  
  
import rclpy  
from rclpy.action import ActionClient  
from rclpy.node import Node  
from my_custom_interfaces.action import MoveTo  
  
class RobotActionClient(Node):  
    def __init__(self):  
        super().__init__('robot_action_client')  
        self._action_client = ActionClient(self, MoveTo, 'move_to')  
  
    def send_goal(self, distance_to_move):  
        goal_msg = MoveTo.Goal()  
        goal_msg.target_distance = distance_to_move  
        self._action_client.wait_for_server()  
        self._send_goal_future = self._action_client.send_goal_async(  
            goal_msg,  
            feedback_callback=self.feedback_callback)  
        self._send_goal_future.add_done_callback(self.  
            ↪ goal_response_callback)  
  
    def goal_response_callback(self, future):  
        goal_handle = future.result()  
        if not goal_handle.accepted:
```

```

        self.get_logger().info('Goal rejected :(')
        return
    self.get_logger().info('Goal accepted :)')
    self.get_result_future = goal_handle.get_result_async()
    self.get_result_future.add_done_callback(self.get_result_callback)

def get_result_callback(self, future):
    result = future.result().result
    self.get_logger().info(f'Goal executed, total distance travelled: {
        ↪ result.total_distance_travelled}m')
    rclpy.shutdown()

def feedback_callback(self, feedback_msg):
    feedback=feedback_msg.feedback
    self.get_logger().info(f'Current distance travelled: {feedback.
        ↪ current_distance_travelled}m')

def main(args=None):
    rclpy.init(args=args)
    robot_action_client = RobotActionClient()
    robot_action_client.send_goal(10.0)
    rclpy.spin(robot_action_client)
    robot_action_client.destroy_node()

if __name__ == '__main__':
    main()

```

---

Listing 1.5. Code listing for robot\_action\_client.py.

#### 4. Build the package

- a. Add the scripts to setup.py under entry\_points:

---

```

entry_points={
    'console_scripts': [
        'robot_action_server=robot_actions.robot_action_server:main',
        'robot_action_client=robot_actions.robot_action_client:main',
        'ra_server_challenge=robot_actions.
            ↪ robot_action_server_challenge:main',
        'ra_client_challenge=robot_actions.
            ↪ robot_action_client_challenge:main',
    ]
}

```

---

Listing 1.6. Include the server and client nodes in the `setup.py` entry points section.

**b. Build the package:**

```
~/Workspaces/ros2_ws$ colcon build
```

---

### 1.5.3 Test the Action Package

1. Open three terminals and source your ROS 2 workspace in each terminal as shown below:

```
~/Workspaces/ros2_ws$ source install/setup.bash
```

2. In Terminal 1, run the Action Server

```
~/Workspaces/ros2_ws$ ros2 run robot_actions robot_action_server
```

3. In Terminal 2, run the Action Client

```
~/Workspaces/ros2_ws$ ros2 run robot_actions robot_action_client
```

4. In Terminal 3, send a goal using the ROS 2 action command

```
~/Workspaces/ros2_ws$ ros2 action send_goal /move_to \  
    my_custom_interfaces/action/MoveTo "{target_distance: 10.0}"
```

Congratulations! You have successfully built and tested a ROS 2 Action system for controlling a robot's movement.

## 1.6 Homework: Implementing Goal Cancellation

For this homework, you will extend your ROS 2 **Action Server** to support goal cancellation. This feature allows an **Action Client** to send a request to cancel a previously sent goal, giving you more control and flexibility over goal execution. When your action server receives a cancel request, it should attempt to stop the goal execution and respond with the appropriate cancellation status. This status will indicate whether the cancellation was successful, rejected, or if the goal was already completed. Adding this functionality will help you better understand how action servers and clients interact, especially when handling dynamic changes.

To help you get started, you can refer to the following ROS 2 examples:

1. [Minimal Action Server](#) – Demonstrates how to set up a goal cancellation callback function on the server side.
2. [Client Goal Cancellation](#) – Shows how to send a cancellation request from the client side and handle the server's response.

Use these examples as a guide to understand how to handle goal cancellation on both the server and client sides. Once you've made the necessary changes, test your implementation by running your action server and client. Verify that the client can send a cancel request and that the server responds with the correct cancellation status.

Take your time to experiment and troubleshoot—this is a great opportunity to deepen your understanding of ROS 2 actions. Good luck, and feel free to ask if you have any questions or run into issues!

## Bibliography