

# Week\_4

## 13장 ROS2 액션

- 공식문서

(<https://docs.ros.org/en/humble/How-To-Guides/Topics-Services-Actions.html#actions>)

# Topics vs Services vs Actions

## Contents

- [Topics](#)
- [Services](#)
- [Actions](#)

When designing a system there are three primary styles of interfaces. The specifications for the content is in the [Interfaces Overview](#). This is written to provide the reader with guidelines about when to use each type of interface.

## Topics

- Should be used for continuous data streams (sensor data, robot state, ...).
- Are for continuous data flow. Data might be published and subscribed at any time independent of any senders/receivers. Many to many connection. Callbacks receive data once it is available. The publisher decides when data is sent.

## Services

- Should be used for remote procedure calls that terminate quickly, e.g. for querying the state of a node or doing a quick calculation such as IK. They should never be used for longer running processes, in particular processes that might be required to preempt if exceptional situations occur and they should never change or depend on state to avoid unwanted side effects for other nodes.
- Simple blocking call. Mostly used for comparably fast tasks as requesting specific data. Semantically for processing requests.

## Actions

- Should be used for any discrete behavior that moves a robot or that runs for a longer time but provides feedback during execution.
- The most important property of actions is that they can be preempted and preemption should always be implemented cleanly by action servers.
- Actions can keep state for the lifetime of a goal, i.e. if executing two action goals in parallel on the same server, for each client a separate state instance can be kept since the goal is uniquely identified by its id.
- Slow perception routines which take several seconds to terminate or initiating a lower-level control mode are good use cases for actions.
- More complex non-blocking background processing. Used for longer tasks like execution of robot actions. Semantically for real-world actions.

# Topics vs Services vs Actions

- system을 디자인할 때에 3가지의 주요한 인터페이스의 스타일이 있다.
- 이러한 콘텐츠에 대한 사항은 Interfaces Overview 문서에 별도로 정리되어 있다.

## Topics

- continuous data streams(센서 데이터, 로봇 상태 정보 등)에 사용하기를 권장하는 인터페이스이다.
- continuous data flow를 위한 인터페이스로, 데이터(Data)는 전송자/발신자 (sender/receiver)관계없이 언제든지 published, subscribe될 수 있다.

## Services

- remote procedure calls that terminate quickly /

빠르게 종료되는(terminate quickly) 원격 프로시저 호출(RPC)를 위해 사용되는 인터페이스이다.

Remote Procedure Call은 컴퓨터 프로그램이 다른 주소 공간에서 Procedure를 실행하도록 하는 것이다(영문 위키 백과,

[https://en.wikipedia.org/wiki/Remote\\_procedure\\_call](https://en.wikipedia.org/wiki/Remote_procedure_call))

- They should never be used for longer running processes, in particular processes **that might be required to preempt if exceptional situations occur** and **they should never change or depend on state to avoid unwanted side effects for other nodes.**

→ 1. 예외적인 상황이 발생할 경우에(**exceptional situations occur**) 서비스는 스레드를 선점(**preempt**)할 수 있도록 설계되어야 한다.

→ 2. 서비스는 다른 노드(other nodes)들의 원치 않는 상태변화를 방지하기 위해서, **로봇의 상태나 의존성을** (change or depend on state) 변경하지 않아야 한다.

- service는 blocking call이다.

↔ topic은 non-blocking call이다. → 결론적으로, 상태를 묻거나(querying the state), 역기구학 [Inverse kinematics (IK)]과 같은 빠른 계산 수행과 같이 빠르게 처리되는 일에만 사용하는 것이 권장된다.

## Actions

- 모든 개별 동작(any discrete behavior)에 사용될 수 있지만, **실행 중에 피드백을 제공**한다.
- 가장 중요한 action의 속성은(The most important property of actions is) action이 **preempted**할 수 있다는 것이다.

ex) 예를 들면, 로봇이 장애물을 앞에 둔 상황에서 action은 스레드를 preempted해서 로봇의 움직임을 멈출 수 있다.

- 액션은 목표(goal)의 수명동안 lifetime을 유지할 수 있다.

ex) 두 개의 action goal이 하나의 서버에서 실행되는 경우, 각각의 goal이 고유한 ID로 인식되기 때문에 각각의 클라이언트에 대해서 별도의 인스턴스 상태를 유지할 수가 있다.

action의 두 가지 예시

**Slow perception routines which take several seconds to terminate** or **initiating a lower-level control mode** are good use cases for actions

- Slow perception routines의 경우
  - 로봇을 주위 환경을 인식시키는 것과 같은 일을 말한다
  - building a map of there surrounding이나 identifying objects와 같은 예시가 있다.
  - 이러한 일들은 **느리게 진행**되고, 완료되는데 **몇 초** 정도가 소모된다.

→ 이러한 **긴 시간을 소모하는 작업**은 한 스레드를 **점유**(preempt)하는 service를 사용하는 것보다 점유하지 않는 action을 사용하는 것이 더 효율적이다.

- Initiating a lower-level control mode
  - 높은 레벨의 알고리즘을 사용하여 로봇을 제어할 경우보다

낮은 레벨에서 로봇을 직접 컨트롤하는 경우에, 피드백을 제공받을 수 있기 때문에 액션을 사용하는 것이 더욱 효과적이다.

## 책 내용

액션(Action)은 비동기/동기식 양방향 메시지 송수신 방식으로

- Goal을 지정하는 Action Client와
- 액션 피드백(Feedback)과 액션 결과(Result)을 전송하는 Action Server 간의 통신이다
- ROS2에서 액션은 topic과 service method의 혼합이라고 볼 수 있는데, Action은 .action 파일에 정의되고
  - 이 파일은 goal / feedback / result에 대한 정의(definition)을 담고 있다.
- 의문 : service는 스레드를 점유하는데, Topic과 Service의 혼합인 Action은 어떻게 비동기일 수 있는가?

→ Action은 topic과 Service의 혼합인것 같은, 유사점은 가지지만 별도의 메시지 형태이다.

- ROS2에서는 send\_goal , cancel\_goal , get\_result를 지원한다.
- **비동기 방식**에서, 원하는 타이밍에 적절한 액션을 수행하지 못하는 문제점을 해결하기 위하여

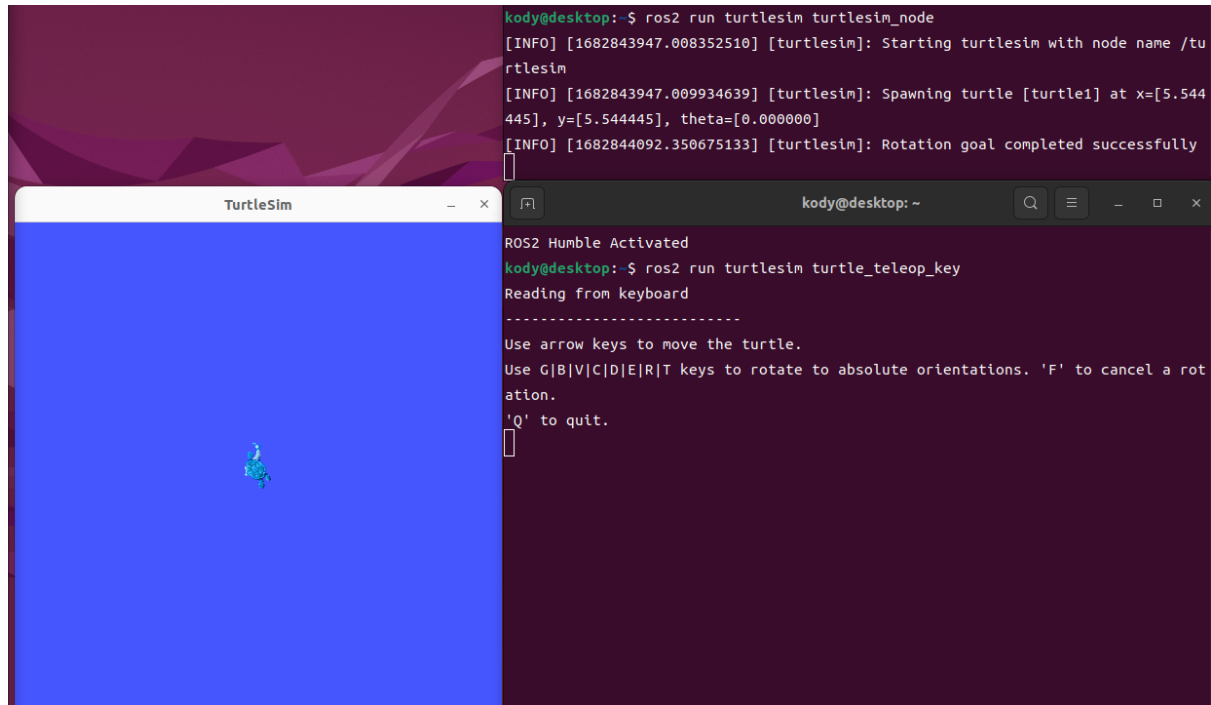
ROS2에서 goal\_state기능을 새롭게 선보였다.

## turtlesim을 이용한 rotate\_absolute 액션 시현

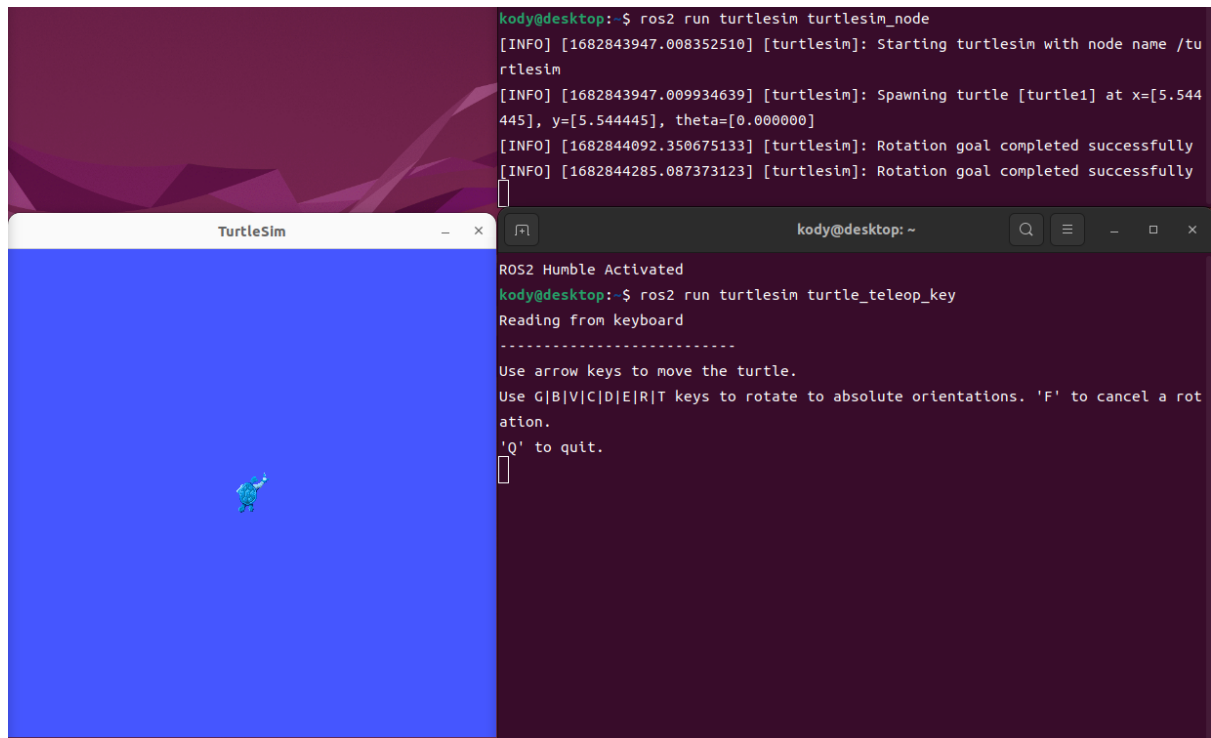
teleop를 시현하면 G ~~~ T를 사용하여 absolute orientation으로 회전시킬 수 있다고 나온다.

G를 입력했을 때에 거북이는 다음 각도로 회전한다.

회전을 완료한 후에는 “Rotation goal completed successfully” 라는 문구가 뜨는 것을 확인할 수 있다.



R키를 입력했을 때에 1.5708radian으로 목표값이 전달되어 거북이가 12시 방향을 바라보는 것을 확인할 수 있다.



## 노드 정보 명령어/액션 목록/액션 정보

### info turtlesim

```
$ ros2 node info /turtlesim
```

```
kody@desktop:~$ ros2 node info /turtlesim
/turtlesim
```

~생략~

```
Action Servers:
  /turtle1/rotate_absolute: turtlesim/action/RotateAbsolute
Action Clients:
```

turtlesim 노드는

- turtlesim/action/RotateAbsolute 인터페이스를 사용하는
- /turtle1/rotate\_absolute 액션 서버를 가지고 있음을 확인할 수 있다.

## info /teleop\_turtle

```
$ ros2 node info /teleop_turtle
```

```
Action Servers:

Action Clients:
  /turtle1/rotate_absolute: turtlesim/action/RotateAbsolute
```

/teleop\_turtle 노드는

- turtlesim/action/RotateAbsolute 인터페이스를 사용하는
- /turtle1/rotate\_absolute 액션 클라이언트를 갖고 있음을 확인할 수 있다.

## rotate\_absolute 액션

- 더 세부적으로, rotate\_absolute 액션은
- send\_goal / cancel\_goal / status / feedback / get\_result를 가지고 있다.

## action list

```
$ ros2 action list -t
```

```
kody@desktop:~$ ros2 action list -t
/turtle1/rotate_absolute [turtlesim/action/RotateAbsolute]
```

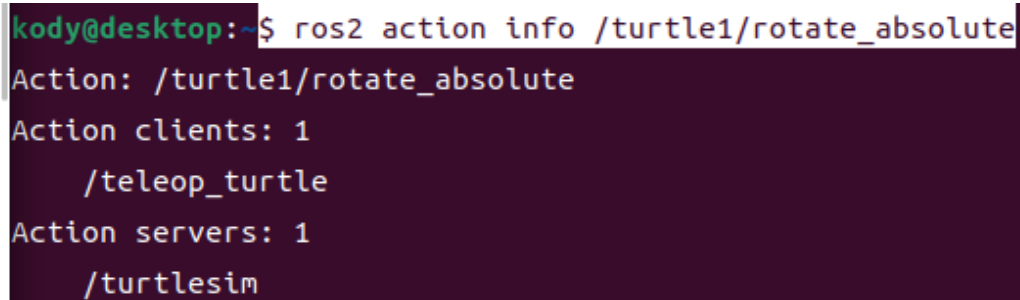


list명령으로

- /turtle1/rotate\_absolute 액션이 실행중이며
- -t 옵션을 추가하여  
[turtlesim/action/RotateAbsolute] 인터페이스를 사용하고 있음을 확인할 수 있다.

## action info

```
$ ros2 action info /turtle1/rotate_absolute
```



```
kody@desktop:~$ ros2 action info /turtle1/rotate_absolute
Action: /turtle1/rotate_absolute
Action clients: 1
    /teleop_turtle
Action servers: 1
    /turtlesim
```

해당 명령에 해당하는 client와 server를 조회하는 명령어이다.

## 액션 목표 전달

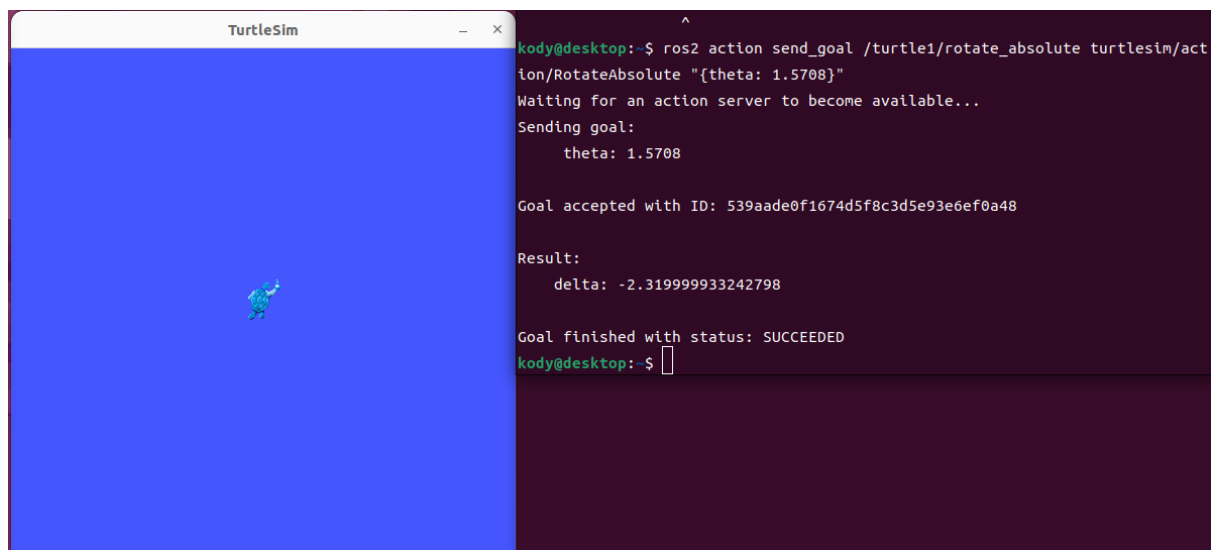
- teleop\_turtle의 Action client를 이용해서도 액션 목표를 전달할 수 있지만,
- ros2 action send\_goal 명령어로도 액션 목표를 전달할 수 있다.

형식

```
$ ros2 action send_goal <action_name> <action_type> "<values>"
```

## 명령

```
$ ros2 action send_goal /turtle1/rotate_absolute  
  turtlesim/action/RotateAbsolute "{theta: 1.5708}"
```



- send\_goal명령어를
  - /turtle1/rotate\_absolute의 액션 이름과
  - turtlesim/action/RotateAbsolute의 형식을 이용하여
  - theta : 1.5708의 Goal을 전달해 주었다.
- 
- 결과값에는
    - goal값
    - 액션 목표의 UID(Unique ID)
    - 결과값
    - 성공 여부

가 표시되는 것을 확인할 수 있다.

## - - feedback 명령 사용

```
ros2 action send_goal /turtle1/rotate_absolute  
turtlesim/action/RotateAbsolute "{theta: 1.5708}"
```

```
Feedback:
    remaining: -0.17320001125335693

Feedback:
    remaining: -0.15720009803771973

Feedback:
    remaining: -0.14120006561279297

Feedback:
    remaining: -0.1252000331878662

Feedback:
    remaining: -0.10920000076293945

Feedback:
    remaining: -0.09320008754730225

Feedback:
    remaining: -0.07720005512237549

Feedback:
    remaining: -0.06120002269744873

Feedback:
    remaining: -0.04520010948181152

Feedback:
    remaining: -0.029200077056884766

Feedback:
    remaining: -0.013200044631958008

Result:
    delta: 2.3359999656677246

Goal finished with status: SUCCEEDED
kody@desktop:~$
```

- Feedback값으로 남은 회전값을 보여준다.

## 14장 ROS2 인터페이스

- ROS2에서 토픽, 서비스, 액션을 주고받을때 사용되는 데이터의 형태를 ROS2 인터페이스라고 한다.

ROS2인터페이스에는

- ROS2에 새롭게 추가된 IDL(Interface Definition Language)와
- msg, srv, action interface등이 있다.

인터페이스 파일의 형식은 다음과 같다

```
fieldtype1 fieldname1
fieldtype2 fieldname2
fieldtype3 fieldname3
```

인터페이스 파일의 예시(vector3.msg파일)

```
float64 x
float64 y
float64 z
```

- ROS2에서 사용 가능한 자료형은 언어별로 약간씩 차이가 있다.

## 메시지 인터페이스

- turtlesim패키지의 경우, /turtle1/cmd\_vel의 토픽은 geometry\_msgs/msg/Twist형태이다.

이 말은, cmd\_vel의 토픽은 geometry\_msgs패키지에 속하는 Twist형태의 데이터를 사용한다는 의미와 같다.

```
$ ros2 interface show geometry_msgs/msg/Twist
```

```
kody@desktop:~$ ros2 interface show geometry_msgs/msg/Twist
# This expresses velocity in free space broken into its linear and angular parts.

Vector3  linear
  float64 x
  float64 y
  float64 z
Vector3  angular
  float64 x
  float64 y
  float64 z
```

Twist파일이 Vector3 linear과 Vector3 angular 에 대한 데이터 형태를 가지고 있음을 알 수 있다.

```
$ ros2 interface show geometry_msgs/msg/Vector3
```

```
kody@desktop:~$ ros2 interface show geometry_msgs/msg/Vector3
# This represents a vector in free space.

# This is semantically different than a point.
# A vector is always anchored at the origin.
# When a transform is applied to a vector, only the rotational component is applied.

float64 x
float64 y
float64 z
```

- 결과적으로, Twist 데이터 형태는 linear.x / linear.y / linear.z / angular.x / angular.y / angular.z

twist.h 파일, 자료형에 대한 부분은 detail/twist\_\_struct.h에 있다.

```
1 // generated from rosidl_generator_c/resource/idl.h.em
2 // with input from geometry_msgs/msg/Twist.idl
3 // generated code does not contain a copyright notice
4
5 #ifndef GEOMETRY_MSGS__MSG__TWIST_H_
6 #define GEOMETRY_MSGS__MSG__TWIST_H_
7
8 #include "geometry_msgs/msg/detail/twist__struct.h"
9 #include "geometry_msgs/msg/detail/twist__functions.h"
10 #include "geometry_msgs/msg/detail/twist__type_support.h"
11
12 #endif // GEOMETRY_MSGS__MSG__TWIST_H_
```

detail/twist\_\_struct.h

열기(O) ▾

twist\_\_struct.h [읽기 전용]

/opt/ros/humble/include/geometry\_msgs/geometry\_msgs/msg/de

```
1 // generated from rosidl_generator_c/resource/idl__struct.h.em
2 // with input from geometry_msgs/msg/Twist.idl
3 // generated code does not contain a copyright notice
4
5 #ifndef GEOMETRY_MSGS__MSG__DETAIL__TWIST__STRUCT_H_
6 #define GEOMETRY_MSGS__MSG__DETAIL__TWIST__STRUCT_H_
7
8 #ifdef __cplusplus
9 extern "C"
10 {
11 #endif
12
13 #include <stdbool.h>
14 #include <stddef.h>
15 #include <stdint.h>
16
17
18 // Constants defined in the message
19
20 // Include directives for member types
21 // Member 'linear'
22 // Member 'angular'
23 #include "geometry_msgs/msg/detail/vector3__struct.h"
24
25 /// Struct defined in msg/Twist in the package geometry_msgs.
26 /**
27  * This expresses velocity in free space broken into its linear and angular parts.
28  */
29 typedef struct geometry_msgs__msg__Twist
30 {
31     geometry_msgs__msg__Vector3 linear;
32     geometry_msgs__msg__Vector3 angular;
33 } geometry_msgs__msg__Twist;
34
35 // Struct for a sequence of geometry_msgs__msg__Twist.
36 typedef struct geometry_msgs__msg__Twist__Sequence
37 {
38     geometry_msgs__msg__Twist * data;
39     /// The number of valid items in data
40     size_t size;
41     /// The number of allocated items in data
42     size_t capacity;
43 } geometry_msgs__msg__Twist__Sequence;
44
45 #ifdef __cplusplus
46 }
47 #endif
48
49 #endif // GEOMETRY_MSGS__MSG__DETAIL__TWIST__STRUCT_H_
```

```
typedef struct geometry_msgs__msg__Twist
{
    geometry_msgs__msg__Vector3 linear;
    geometry_msgs__msg__Vector3 angular;
} geometry_msgs__msg__Twist;
```



linear과 angular정보는 geometry\_msgs\_\_msg\_\_Vector3의 형태로 정의되어 있는 것을 확인할 수 있다.

- ros2의 인터페이스 조회

```
kody@desktop:~$ ros2 interface list
Messages:
  action_msgs/msg/GoalInfo
  action_msgs/msg/GoalStatus
  action_msgs/msg/GoalStatusArray
  actionlib_msgs/msg/GoalID
  actionlib_msgs/msg/GoalStatus
  actionlib_msgs/msg/GoalStatusArray
  builtin_interfaces/msg/Duration
  builtin_interfaces/msg/Time
  diagnostic_msgs/msg/DiagnosticArray
  diagnostic_msgs/msg/DiagnosticStatus
  diagnostic_msgs/msg/KeyValue
  example_interfaces/msg/Bool
  example_interfaces/msg/Byte
  example_interfaces/msg/ByteMultiArray
  example_interfaces/msg/Char
  example_interfaces/msg/Empty
  example_interfaces/msg/Float32
  example_interfaces/msg/Float32MultiArray
  example_interfaces/msg/Float64
  example_interfaces/msg/Float64MultiArray
  example_interfaces/msg/Int16
  example_interfaces/msg/Int16MultiArray
  example_interfaces/msg/Int32
  example_interfaces/msg/Int32MultiArray
  example_interfaces/msg/Int64
  example_interfaces/msg/Int64MultiArray
  example_interfaces/msg/Int8
  example_interfaces/msg/Int8MultiArray
  example_interfaces/msg/MultiArrayDimension
  example_interfaces/msg/MultiArrayLayout
  example_interfaces/msg/String
  example_interfaces/msg/UInt16
  example_interfaces/msg/UInt16MultiArray
  example_interfaces/msg/UInt32
  example_interfaces/msg/UInt32MultiArray
  example_interfaces/msg/UInt64
  example_interfaces/msg/UInt64MultiArray
  example_interfaces/msg/UInt8
  example_interfaces/msg/UInt8MultiArray
  example_interfaces/msg/WString
  geometry_msgs/msg/Accel
  geometry_msgs/msg/AccelStamped
  geometry_msgs/msg/AccelWithCovariance
  geometry_msgs/msg/AccelWithCovarianceStamped
  geometry_msgs/msg/Inertia
  geometry_msgs/msg/InertiaStamped
  geometry_msgs/msg/Point
```

- ros2의 인터페이스 패키지 조회

```
kody@desktop:~$ ros2 interface packages
action_msgs
action_tutorials_interfaces
actionlib_msgs
builtin_interfaces
composition_interfaces
diagnostic_msgs
example_interfaces
geometry_msgs
lifecycle_msgs
logging_demo
map_msgs
nav_msgs
pcl_msgs
pendulum_msgs
rcl_interfaces
rmw_dds_common
rosbag2_interfaces
rosgraph_msgs
sensor_msgs
shape_msgs
statistics_msgs
std_msgs
std_srvs
stereo_msgs
tf2_msgs
trajectory_msgs
turtlesim
unique_identifier_msgs
visualization_msgs
```

- ros2의 turtlesim 패키지가 사용하는 인터페이스 조회

```
kody@desktop:~$ ros2 interface package turtlesim
turtlesim/msg/Color
turtlesim/msg/Pose
turtlesim/srv/Kill
turtlesim/action/RotateAbsolute
turtlesim/srv/TeleportAbsolute
turtlesim/srv/Spawn
turtlesim/srv/SetPen
turtlesim/srv/TeleportRelative
```

- ros2 interface proto ~ 명령을 이용하여 자료형을 출력

```
kody@desktop:~$ ros2 interface proto geometry_msgs/msg/Twist
"linear:
  x: 0.0
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 0.0
"
```

## ROS2 서비스 인터페이스

- 서비스에 대한 인터페이스 파일은 .srv 파일 안에 정의한다.

```
$ ros2 interface show turtlesim/srv/Spawn
```

```
kody@desktop:~$ ros2 interface show turtlesim/srv/Spawn
float32 x
float32 y
float32 theta
string name # Optional. A unique name will be created and returned if this is empty
---
string name
```

- 이 메시지가 x,y,theta값과 name을 갖고 있음을 알 수 있고,
- - - - 위에 있는 string name은 Request이고 아래는 Response이다.

## 액션 인터페이스

```
$ ros2 interface show turtlesim/action/RotateAbsolute
```

```
kody@desktop:~$ ros2 interface show turtlesim/action/RotateAbsolute
# The desired heading in radians
float32 theta
---
# The angular displacement in radians to the starting position
float32 delta
---
# The remaining rotation in radians
float32 remaining
```

.action 파일의 내용이다.

- theta 값은 액션 목표이다
- delta 값은 액션 결과이다
- remaining 값은 액션 피드백을 나타낸다.

## 15장 Topic/Service/Action 정리 비교

- 각각의 특징 비교

	토픽	서비스	액션
연속성	연속적	일회적	일회적
방향성	단방향	양방향	양방향

동기성	비동기	동기	비동기 + 동기
노드 역할	Publisher : Subscriber	Server : Client	Server : Client
인터페이스	msg인터페이스	srv인터페이스	action인터페이스
명령어	ros2 topic / ros2 interface	ros2 service / ros2 interface	ros2 action / ros2 interface

- 인터페이스 비교

## msg인터페이스

- \*.msg
- 데이터 : data
- 형식
  - fieldtype1 fieldname1
  - fieldtype2 fieldname2
  - fieldtype3 fieldname3
- 사용예
  - [geometry\_msgs/msg/Twist]
  - Vector3 linear
  - Vector3 angular

## srv인터페이스

- \*.srv
- 데이터
  - request
  - - - -
  - response
- 형식

- fieldType1 fieldName1
- fieldType2 fieldName2
- - - -
- fieldType3 fieldName3
- fieldType4 fieldName4
- 사용예
  - [turtlesim/srv/Spawn.srv]
  - float32 x
  - float32 y
  - float32 theta
  - string name
  - - - -
  - string name

## action인터페이스

- \*.action
- 데이터
  - action goal
  - - - -
  - action result
  - - - -
  - action feedback
- 형식
  - fieldType1 fieldName1
  - fieldType2 fieldName2
  - - - -
  - fieldType3 fieldName3

- fieldType4 fieldname4
- - - -
- fieldType5 fieldname5
- fieldType6 fieldname6
- 사용예
  - float32 theta
  - - - -
  - float32 delta
  - - - -
  - float32 remaining

## 16장 파라미터

공식문서

- **About parameters in ROS 2**

(<https://docs.ros.org/en/humble/Concepts/About-ROS-2-Parameters.html#parameter-types>)

책의 내용

- ros2에서 파라미터는 노드 내의 매개변수로,
- 각 노드는 parameter server를 가지고 있고, 외부의 parameter client와 통신하여 파라미터를 변경할 수 있다.
- 모든 노드는 parameter server과 parameter client를 가질 수 있으며, 이를 활용하여 다양한 매개변수를 글로벌 매개변수 처럼 활용할 수 있다.

```
$ ros2 run turtlesim turtlesim_node
```



```
$ ros2 run turtlesim turtle_teleop_key
```

## 파라미터 목록 확인(ros2 param list)

```
$ ros2 param list
```

## 파라미터 내용 확인(ros2 param describe)

```
$ ros2 param describe /turtlesim background_b
```

→ parameter name :

Type, Description, Constraints(Min, Max, Step)

## 파라미터 읽기(ros2 param get)

- 형식

```
$ ros2 param get <node_name> <parameter_name>
```

→ <Type> value is :

## 파라미터 쓰기(ros2 param set)

- 형식

```
$ ros2 param set <node_name> <parameter_name> <value>
```

→ Set parameter successful

## 파라미터 저장(ros2 param dump)

- ros2 param dump는 (turtlesim)노드를 종료한 후에 다시 실행할 경우에 파라미터 값이 초기화되며, 이 경우에 파라미터 저장 명령어(ros2 param dump)를 이용하여 변경된 파라미터를 저장할 수 있다.

- 형식

```
$ ros2 param dump /turtlesim
```

→ Saving to: ./turtlesim.yaml

실행하면, 현재 경로에 설정 파일이 .yaml 형식으로 저장된다.

- turtlesim.yaml 출력

```
$ cat ./turtlesim.yaml
```

ros\_\_parameter:

~~~

- 노드 실행시 —ros -args —params-file 옵션을 사용하여 yaml 파일 위치를 지정하면 저장된 파라미터로 지정해 줄 수 있다.

```
$ ros2 run turtlesim turtlesim_node --ros-args --params-file ./turtlesim.yaml
```

## 파라미터 삭제(ros2 param delete)

- 형식

```
$ ros2 param delete <node_name> <parameter_name>
```

- 명령어

```
$ ros2 param delete /turtlesim background_b
```

→ Delete parameter successfully