ROS2 Foundations
Day 2
Visualization, Launch Nodes, Services & Actions

From the Visualization in ROS2 to Using Launch Files for Topics, Services and Actions

Thomas Birchler

UEB

URDF vs Xacro

URDF (Unified Robot Description Format)

- XML format describing robot links, joints, sensors, visuals, and collisions.
- Good for static robots, hard to reuse or parametrize.
- Used in: RViz, Gazebo, TF2, MoveIt

Xacro (XML Macros)

- Adds macros, variables, conditionals to URDF.
- Write DRY code: reuse repeated parts, easily generate variants.
- Process with: `ros2 run xacro xacro mybot.xacro > mybot.urdf`

URDF Example Snippet 1/2

URDF example snippet

```
<robot name="my_robot">
  <link name="base_link">
    <visual>
      <geometry>
        <box size="1 1 1"/>
      </geometry>
    </visual>
  </link>

  <joint name="joint1" type="continuous">
    <parent link="base_link"/>
    <child link="wheel_link"/>
    <origin xyz="0 1 0" rpy="0 0 0"/>
    <axis xyz="0 0 1"/>
  </joint>
```

URDF Example Snippet 2/2

URDF example snippet

```
  <link name="wheel_link">
    <visual>
      <geometry>
        <cylinder radius="0.2" length="0.05"/>
      </geometry>
    </visual>
  </link>
</robot>
```

URDF/Xacro Anatomy

Core elements:

- In Xacro: you can define arguments, properties, macros

Xacro example snippet

```
<xacro:property name="wheel_radius" value="0.05"/>
<xacro:macro name="wheel" params="name x y">
```

Xarco Example Snippet 1/2

Xarco example snippet

```
<?xml version="1.0"?>
<robot xmlns:xacro="http://ros.org/wiki/xacro"
    name="xacro_robot">

  <!-- Parameters -->
  <xacro:property name="wheel_radius" value="0.2"/>
  <xacro:property name="wheel_thickness" value="0.05"/>
```

Xarco Example Snippet 2/3

Xarco example snippet

```
<!-- Macro -->
  <xacro:macro name="wheel" params="name x y">
    <link name="${name}_link">
      <visual>
        <geometry>
          <cylinder radius="${wheel_radius}" length=
            "${wheel_thickness}"/>
        </geometry>
        <origin xyz="${x} ${y} 0"/>
      </visual>
    </link>
  </xacro:macro>
```

Xarco Example Snippet 3/3

Xarco example snippet

```
  <!-- Use macro -->
  <xacro:wheel name="front_left" x="1" y="1"/>
  <xacro:wheel name="front_right" x="1" y="-1"/>
  <xacro:wheel name="rear_left" x="-1" y="1"/>
  <xacro:wheel name="rear_right" x="-1" y="-1"/>

</robot>
```

URDF vs Xacro – Summary

Summary:

- **URDF (Unified Robot Description Format):** static, no support for variables, loops, or reuse
- **Xacro (XML Macros):** Enables reuse and easier parameter tuning.

Typical Workflow:

- Write robot model as `.xacro` file.
- Launch using RViz or Gazebo via launch file (`launch/view_robot.launch.py`).
- Maintain only `.xacro`; generate `.urdf` if needed for inspection.

1/2 Steps to Create a Xacro-based Robot Package

Minimal ROS 2 Package Setup (Xacro + Launch)

- **Workspace + Package**

```
source /opt/ros/humble/setup.bash
mkdir -p ~/ros2_ws/src && cd ~/ros2_ws/src
ros2 pkg create --build-type ament_cmake my_robot_description
```

- **URDF and Launch Setup**

```
cd my_robot_description && mkdir urdf launch
nano urdf/my_robot.xacro      # paste your robot model
touch launch/robot_display.launch.py
```

- **CMakeLists.txt**

```
find_package(ament_cmake REQUIRED)
install(DIRECTORY urdf/ DESTINATION share/${PROJECT_NAME}/urdf
install(DIRECTORY launch/ DESTINATION share/${PROJECT_NAME}/la
ament_package()
```

2/2 Steps to Create a Xacro-based Robot Package

- **package.xml**

```
<exec_depend>xacro</exec_depend>
<exec_depend>robot_state_publisher</exec_depend>
```

- **Build + Launch**

```
cd ~/ros2_ws && colcon build
source install/setup.bash
ros2 launch my_robot_description robot_display.launch.py
```

RViz2 – 3D Visualization Tool

What is RViz2?

- 3D visualization tool for ROS 2 data streams.
- Visualizes robot state, sensor data, coordinate frames, and environment.
- Helps you debug and understand what your robot sees and does.

Common Uses:

- View URDF robot models and TF tree.
- Plot sensor data like LIDAR, cameras, IMUs.
- Inspect published topics (e.g., velocity commands, paths).
- Set navigation goals or markers interactively.

Launch Example:

```
ros2 run rviz2 rviz2
```

Gazebo – Simulation Environment for ROS 2

What is Gazebo?

- Open-source 3D simulator with physics engine for robotics.
- Used to test robot models, sensors, and algorithms in a realistic virtual environment.

Key Features:

- Physics simulation (gravity, collisions, inertia).
- Sensor simulation (camera, LIDAR, IMU).
- Plugin interface for actuators, controllers, and custom behavior.
- Supports URDF/Xacro-based robot models.

Common Uses

- Validate robot designs and code before deploying to real hardware.
- Simulate complex environments with obstacles and terrain.
- Run automated experiments and integration tests in a reproducible way.

Gazebo vs. RViz - When to Use Which Tool?

- Use **Gazebo** for simulating the real world
- Use **RViz** to inspect and debug what's happening inside your robot.

Feature	**Gazebo**	**RViz**
Purpose	Full physics-based simulation	Visualization of sensor data and frames
Physics engine	✓Gravity, collisions, dynamics	✗ No physics
Sensor simulation	✓Virtual camera, LIDAR, IMU	✗ Only displays existing data
Robot control testing	✓Simulate motion, actuators	✗ Read-only view
TF frames	✓Publishes frames	✓Visualizes TF tree
Typical use	Test control	Debug data, perception, TF
Performance	Heavier (sim engine)	Lightweight

Launch Files – Why and How?

What are Launch Files?

- Launch files automate the startup of multiple ROS 2 nodes.
- Written in Python using `launch` and `launch_ros` libraries.

Why Use Launch Files?

- Manual startup of nodes is tedious and error-prone.
- Encapsulate complex startup logic in one place.
- Reuse across simulations, hardware tests, and deployments.

Typical Use Case:

- Start a robot description node (URDF/Xacro)
- Launch a controller (e.g. diff drive plugin)
- Bring up RViz or Gazebo with matching configuration

Run a Launch File

```
ros2 launch my_package my_launch_file.py
```

Launch Files - Example

Python-based launch system

- Start multiple nodes with shared config.
- Set parameters, remappings, names, namespaces.

Example

```
from launch import LaunchDescription
from launch_ros.actions import Node

def generate_launch_description():
return LaunchDescription([
Node(
package='my_robot',
executable='talker',
parameters=[{'rate': 10.0}]
)
])
```

Service Node – Explained

Key Ideas:

- A **service server** waits for requests on a named service.
- It uses a **callback function** to handle incoming requests.
- Each request triggers the callback exactly once, with a response returned.

Key Methods:

- `create_service` sets up the service and callback.
- `request` contains input data from the client.
- `return response` sends data back to the client.

Key Implementation:

- Define the service interface in a dedicated package.
- Implement the service logic in a separate (e.g., functional) package.

1/10 Service - Create Interface

- It is highly recommended to place service interface definitions in a dedicated package.
- The implementation of the corresponding service node can reside in a separate package.

Create new package in /src

```
ros2 pkg create \
  --build-type ament_cmake \
  radio_communication_solutions_interfaces
```

2/10 Service - Create Interface

- Create file that defines the interface of request and response message.
- Location: <package_name_service>/srv/<service_name>

srv/GetNowPlaying.srv

```
# Request (none needed in our specific case)
---


# Response
string station_name
string song_title
string artist_name
string time_played
```

3/10 Service - Create Interface

Replace in CMakelists.txt with the following

```
cmake_minimum_required(VERSION 3.5)
project(<radio_communication_interfaces>)

find_package(ament_cmake REQUIRED)
find_package(rosidl_default_generators REQUIRED)

rosidl_generate_interfaces(${PROJECT_NAME}
  "srv/<GetNowPlaying.srv>"
)

ament_export_dependencies(rosidl_default_runtime)
ament_package()
```

Use respective srv name for <GetNowPlaying.srv>.

4/10 Service - Node Creation

Add following lines to package.xml

```xml
<buildtool_depend>ament_cmake</buildtool_depend>

<build_depend>rosidl_default_generators</build_depend>
<exec_depend>rosidl_default_runtime</exec_depend>

<member_of_group>rosidl_interface_packages</member_of_group>
```

5/10 Service - Node Creation

Add dependencies into package.xml

```
<exec_depend>package_name_interface</exec_depend>
```

In our case use radio_communication_interfaces.
Hint: Even if a package is typically available, it's good practice to declare it explicitly to avoid issues in other environments. (Common packages are: rclpy and std_msgs))

Add entry point and data files in setup.py

```
data_files=[
    (os.path.join('share', package_name, 'srv'),
        glob('srv/*.srv')),
],
entry_points={
    'console_scripts': [
        'get_current_song_service =
            radio_communication.get_current_song_service:main',
    ],
},
```

7/10 Service - Node Creation

get_now_playing_service.py, class <Service>(Node)

```
from <GetNowPlaying.srv> import <GetNowPlaying>

class GetNowPlaying(Node):
    def __init__(self):
        super().__init__('get_now_playing')
        self.srv = self.create_service(
            <Service Data Type>,
            '<Service Name>',
            self.<callback_function>
            )
```

Replace <> with:
GetNowPlaying, 'get_now_playing', handle_now_playing

8/10 Service - Node Creation

get_now_playing_service.py, class <Service>(Node)

```
def handle_now_playing(self, request, response):
        response.station_name = "102 FM"
        response.song_title = "Bohemian Rhapsody"
        response.artist_name = "Queen"
        response.time_played =
            datetime.now().strftime("%H:%M:%S")
        self.get_logger().info("Returned current track info.")
        return response
```

9/10 Service - Node Creation

get_now_playing_service.py, main()

```
def main(args=None):
    rclpy.init(args=args)
    node = SongServiceNode()
    rclpy.spin(node)
    rclpy.shutdown()

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

10/10 Service - Node Creation

Build the package

```
colcon build --packages-select radio_communication
source install/setup.bash
```

Action Server – Explained

Key Ideas:

- An **action server** accepts long-running goals from clients.
- It provides **feedback during execution** and a final **result**.
- Clients may cancel or get status updates during execution.

Key Methods:

- `create_action_server` sets up the action and callbacks.
- `goal_callback` determines whether to accept a goal.
- `execute_callback` performs the action logic and provides feedback.
- `result` is returned once the goal completes.

Key Implementation:

- Define the `.action` interface in a dedicated interface package.
- Use `rclpy.action.ActionServer` for Python or `rclcpp::ActionServer` in C++.
- Handle feedback and cancellation logic within the execution callback.

1/8 Action Creation – Interface Package

Create package + directory

```
ros2 pkg create <pkg_name_interfaces> --build-type ament_cmake
cd <pkg_name_interfaces>
mkdir action
```

Replace <pkg_name_interfaces> with radio_station_interfaces
Include inside package:

- `action/ChooseSong.action`
- Updated `CMakeLists.txt` (+ `rosidl_generate_interfaces`)
- Updated `package.xml` (see slide 3/8)

2/8 Action Creation - Interface Package

action/ChooseSong.action

```
# Goal: what caller tells DJ
string song_name
---
# Result: final DJ reply
bool   success
string final_message
---
# Feedback: in-progress updates
float32 progress_pct
string  status_line
```

3/8 Action Creation - Interface Package

CMakeLists.txt essentials

```
find_package(ament_cmake REQUIRED)
find_package(rosidl_default_generators REQUIRED)

rosidl_generate_interfaces(${PROJECT_NAME}
  "action/ChooseSong.action"
)
ament_package()
```

package.xml additions

```
<buildtool_depend>ament_cmake</buildtool_depend>
<build_depend>rosidl_default_generators</build_depend>
<exec_depend>rosidl_default_runtime</exec_depend>

<member_of_group>rosidl_interface_packages</member_of_group>
```

4/8 Action Creation - Build Interfaces

One-time build

```
cd ~/workspace
colcon build --packages-select radio_station_interfaces
source install/setup.bash
```

Result: generated headers & Python modules for `ChooseSong`.

Create Python package + deps

```
ros2 pkg create <package_name> --build-type ament_python \
  --dependencies rclpy <package_name_interfaces>
```

Replace <package_name> with radio_station and <package_name_interfaces> with radio_station_interfaces.

Add entry point and data files in setup.py

```
entry_points={
    'console_scripts': [
        'radio_dj = radio_station.server_node:main',
    ],
},
```

6/8 Action Creation – Server Node

radio_station/server_node.py (class RadioDJ(Node))

```
def __init__(self):
    super().__init__('radio_dj')
    self._srv = ActionServer(
        self, ChooseSong, 'choose_song', self.execute_cb)

async def execute_cb(self, goal_handle):
    song = goal_handle.request.song_name
    for pct in range(0, 101, 20):
        fb = ChooseSong.Feedback(progress_pct=float(pct),
                                 status_line=f"Playing {song}")
        goal_handle.publish_feedback(fb)
        await asyncio.sleep(1)
    res = ChooseSong.Result(success=True,
                            final_message=f"Finished {song}")
    return res
```

7/8 Action Creation - Client Node 1/4

setup.py – second entry point

```
'console_scripts': [
    'radio_dj = radio_station.server_node:main',
    'song_requester = radio_station.client_node:main',
],
```

radio_station/client_node.py (core logic)

```
goal = ChooseSong.Goal(song_name="Bohemian Rhapsody")
client.send_goal_async(goal,
    feedback_callback=lambda fb:
      node.get_logger().info(f"DJ: {fb.feedback.status_line}"))
```

7/8 Action Creation - Client Node 2/4

radio_station/client_node.py: class SongRequester(Node)

```
def __init__(self, song):
    super().__init__('song_requester')
    self._action_client = ActionClient(self, ChooseSong,
        'choose_song')
    self._song = song
    self._action_client.wait_for_server()
    self.send_request()

def send_request(self):
    goal = ChooseSong.Goal(song_name=self._song)
    self._send_future = self._action_client.send_goal_async(
        goal,
        feedback_callback=self.feedback_cb)
    self._send_future.add_done_callback(self.goal_response_cb)
```

7/8 Action Creation - Client Node 3/4

radio_station/client_node.py: class SongRequester(Node)

```
def feedback_cb(self, feedback_msg):
    fb = feedback_msg.feedback
    self.get_logger().info(f"DJ says: {fb.status_line}")

def goal_response_cb(self, future):
    goal_handle = future.result()
    result_future = goal_handle.get_result_async()
    result_future.add_done_callback(self.result_cb)

def result_cb(self, future):
    result = future.result().result
    self.get_logger().info(f"DJ final: {result.final_message}")
    rclpy.shutdown()
```

7/8 Action Creation - Client Node 4/4

radio_station/client_node.py

```
def main():
    rclpy.init()
    requester = SongRequester("Bohemian Rhapsody")
    rclpy.spin(requester)

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

8/8 Running the Radio Request Demo

Build everything

```
colcon build
source install/setup.bash
```

Run:

1. Terminal 1 (DJ): `ros2 run radio_station radio_dj`
2. Terminal 2 (Caller): `ros2 run radio_station song_requester`

Watch the feedback stream and final message – you just "called the station and chose a song" via a ROS 2 Action!