You're reading the documentation for an older, but still supported, version of ROS 2. For information on the latest version, please have a look at Iron.

Creating custom msg and srv files

Goal: Define custom interface files (`.msg` and `.srv`) and use them with Python and C++ nodes.

Tutorial level: Beginner

Time: 20 minutes

Contents

Background

In previous tutorials you utilized message and service interfaces to learn about topics, services, and simple publisher/subscriber (C++/Python) and service/client (C++/Python) nodes. The interfaces you used were predefined in those cases.

While it's good practice to use predefined interface definitions, you will probably need to define your own messages and services sometimes as well. This tutorial will introduce you to the simplest method of creating custom interface definitions.

Prerequisites

You should have a ROS 2 workspace.

This tutorial also uses the packages created in the publisher/subscriber (C++ and Python) and service/client (C++ and Python) tutorials to try out the new custom messages.

Tasks

1 Create a new package

For this tutorial you will be creating custom `.msg` and `.srv` files in their own package, and then utilizing them in a separate package. Both packages should be in the same workspace.

Since we will use the pub/sub and service/client packages created in earlier tutorials, make sure you are in the same workspace as those packages (`ros2_ws/src`), and then run the following command to create a new package:

```
ros2 pkg create --build-type ament_cmake --license Apache-2.0 tutorial_interfaces
```

`tutorial_interfaces` is the name of the new package. Note that it is, and can only be, a CMake package, but this doesn't restrict in which type of packages you can use your messages and services. You can create your own custom interfaces in a CMake package, and then use it in a C++ or Python node, which will be covered in the last section.

The `.msg` and `.srv` files are required to be placed in directories called `msg` and `srv` respectively. Create the directories in `ros2_ws/src/tutorial_interfaces`:

```
mkdir msg srv
```

2 Create custom definitions

2.1 msg definition

In the `tutorial_interfaces/msg` directory you just created, make a new file called `Num.msg` with one line of code declaring its data structure:

```
int64 num
```

This is a custom message that transfers a single 64-bit integer called `num`.

Also in the `tutorial_interfaces/msg` directory you just created, make a new file called `Sphere.msg` with the following content:

```
geometry_msgs/Point center
float64 radius
```

This custom message uses a message from another message package (`geometry_msgs/Point` in this case).

2.2 srv definition

Back in the `tutorial_interfaces/srv` directory you just created, make a new file called `AddThreeInts.srv` with the following request and response structure:

```
int64 a
int64 b
int64 c
---
int64 sum
```

This is your custom service that requests three integers named `a`, `b`, and `c`, and responds with an integer called `sum`.

3 `CMakeLists.txt`

To convert the interfaces you defined into language-specific code (like C++ and Python) so that they can be used in those languages, add the following lines to `CMakeLists.txt`:

```
find_package(geometry_msgs REQUIRED)
find_package(rosidl_default_generators REQUIRED)

rosidl_generate_interfaces(${PROJECT_NAME}
  "msg/Num.msg"
  "msg/Sphere.msg"
  "srv/AddThreeInts.srv"
  DEPENDENCIES geometry_msgs # Add packages that above messages depend on, in this case
geometry_msgs for Sphere.msg
)
```

Note

The first argument (library name) in the rosidl_generate_interfaces must match ${PROJECT_NAME} (see https://github.com/ros2/rosidl/issues/441#issuecomment-591025515).

4 `package.xml`

Because the interfaces rely on `rosidl_default_generators` for generating language-specific code, you need to declare a build tool dependency on it. `rosidl_default_runtime` is a runtime or execution-stage dependency, needed to be able to use the interfaces later. The `rosidl_interface_packages` is the name of the dependency group that your package, `tutorial_interfaces`, should be associated with, declared using the `<member_of_group>` tag.

Add the following lines within the `<package>` element of `package.xml`:

```
<depend>geometry_msgs</depend>
<buildtool_depend>rosidl_default_generators</buildtool_depend>
<exec_depend>rosidl_default_runtime</exec_depend>
<member_of_group>rosidl_interface_packages</member_of_group>
```

5 Build the `tutorial_interfaces` package

Now that all the parts of your custom interfaces package are in place, you can build the package. In the root of your workspace (`~/ros2_ws`), run the following command:

Linux | macOS | Windows

```
colcon build --packages-select tutorial_interfaces
```

Now the interfaces will be discoverable by other ROS 2 packages.

6 Confirm msg and srv creation

In a new terminal, run the following command from within your workspace (`ros2_ws`) to source it:

Linux | macOS | Windows

```
source install/setup.bash
```

Now you can confirm that your interface creation worked by using the `ros2 interface show` command:

```
ros2 interface show tutorial_interfaces/msg/Num
```

should return:

```
int64 num
```

And

```
ros2 interface show tutorial_interfaces/msg/Sphere
```

should return:

```
geometry_msgs/Point center
        float64 x
        float64 y
        float64 z
float64 radius
```

And

```
ros2 interface show tutorial_interfaces/srv/AddThreeInts
```

should return:

```
int64 a
int64 b
int64 c
---
int64 sum
```

7 Test the new interfaces

For this step you can use the packages you created in previous tutorials. A few simple modifications to the nodes, `CMakeLists.txt` and `package.xml` files will allow you to use your new interfaces.

7.1 Testing `Num.msg` with pub/sub

With a few modifications to the publisher/subscriber package created in a previous tutorial (C++ or Python), you can see `Num.msg` in action. Since you'll be changing the standard string msg to a numerical one, the output will be slightly different.

Publisher

C++ | Python

```cpp
#include <chrono>
#include <memory>

#include "rclcpp/rclcpp.hpp"
#include "tutorial_interfaces/msg/num.hpp"                                    //
CHANGE

using namespace std::chrono_literals;

class MinimalPublisher : public rclcpp::Node
{
public:
  MinimalPublisher()
  : Node("minimal_publisher"), count_(0)
  {
    publisher_ = this->create_publisher<tutorial_interfaces::msg::Num>("topic", 10);  //
CHANGE
    timer_ = this->create_wall_timer(
      500ms, std::bind(&MinimalPublisher::timer_callback, this));
  }

private:
  void timer_callback()
  {
    auto message = tutorial_interfaces::msg::Num();                           //
CHANGE
    message.num = this->count_++;                                             //
CHANGE
    RCLCPP_INFO_STREAM(this->get_logger(), "Publishing: '" << message.num << "'");    //
CHANGE
    publisher_->publish(message);
  }
  rclcpp::TimerBase::SharedPtr timer_;
  rclcpp::Publisher<tutorial_interfaces::msg::Num>::SharedPtr publisher_;     //
CHANGE
  size_t count_;
};

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

Subscriber

C++ | Python

```
#include <functional>
#include <memory>

#include "rclcpp/rclcpp.hpp"
#include "tutorial_interfaces/msg/num.hpp"                                      // CHANGE

using std::placeholders::_1;

class MinimalSubscriber : public rclcpp::Node
{
public:
  MinimalSubscriber()
  : Node("minimal_subscriber")
  {
    subscription_ = this->create_subscription<tutorial_interfaces::msg::Num>(    // CHANGE
      "topic", 10, std::bind(&MinimalSubscriber::topic_callback, this, _1));
  }

private:
  void topic_callback(const tutorial_interfaces::msg::Num & msg) const  // CHANGE
  {
    RCLCPP_INFO_STREAM(this->get_logger(), "I heard: '" << msg.num << "'");     // CHANGE
  }
  rclcpp::Subscription<tutorial_interfaces::msg::Num>::SharedPtr subscription_;  // CHANGE
};

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

CMakeLists.txt

Add the following lines (C++ only):

```
#...

find_package(ament_cmake REQUIRED)
find_package(rclcpp REQUIRED)
find_package(tutorial_interfaces REQUIRED)                      # CHANGE

add_executable(talker src/publisher_member_function.cpp)
ament_target_dependencies(talker rclcpp tutorial_interfaces)    # CHANGE

add_executable(listener src/subscriber_member_function.cpp)
ament_target_dependencies(listener rclcpp tutorial_interfaces)  # CHANGE

install(TARGETS
  talker
  listener
  DESTINATION lib/${PROJECT_NAME})

ament_package()
```

package.xml

Add the following line:

C++ | Python

```
<depend>tutorial_interfaces</depend>
```

After making the above edits and saving all the changes, build the package:

C++ | Python

On Linux/macOS:

```
colcon build --packages-select cpp_pubsub
```

On Windows:

```
colcon build --merge-install --packages-select cpp_pubsub
```

Then open two new terminals, source `ros2_ws` in each, and run:

C++ | Python

```
ros2 run cpp_pubsub talker
```

```
ros2 run cpp_pubsub listener
```

Since `Num.msg` relays only an integer, the talker should only be publishing integer values, as opposed to the string it published previously:

```
[INFO] [minimal_publisher]: Publishing: '0'
[INFO] [minimal_publisher]: Publishing: '1'
[INFO] [minimal_publisher]: Publishing: '2'
```

7.2 Testing `AddThreeInts.srv` with service/client

With a few modifications to the service/client package created in a previous tutorial (C++ or Python), you can see `AddThreeInts.srv` in action. Since you'll be changing the original two integer request srv to a three integer request srv, the output will be slightly different.

Service

C++ | Python

```
#include "rclcpp/rclcpp.hpp"
#include "tutorial_interfaces/srv/add_three_ints.hpp"
// CHANGE

#include <memory>

void add(const std::shared_ptr<tutorial_interfaces::srv::AddThreeInts::Request> request,
// CHANGE
          std::shared_ptr<tutorial_interfaces::srv::AddThreeInts::Response>       response)
// CHANGE
{
  response->sum = request->a + request->b + request->c;
// CHANGE
  RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "Incoming request\na: %ld" " b: %ld" " c: %ld",
// CHANGE
                request->a, request->b, request->c);
// CHANGE
  RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "sending back response: [%ld]", (long
int)response->sum);
}

int main(int argc, char **argv)
{
  rclcpp::init(argc, argv);

  std::shared_ptr<rclcpp::Node> node = rclcpp::Node::make_shared("add_three_ints_server");
// CHANGE

  rclcpp::Service<tutorial_interfaces::srv::AddThreeInts>::SharedPtr service =
// CHANGE
    node->create_service<tutorial_interfaces::srv::AddThreeInts>("add_three_ints",  &add);
// CHANGE

  RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "Ready to add three ints.");
// CHANGE

  rclcpp::spin(node);
  rclcpp::shutdown();
}
```

Client

C++ | Python

```
#include "rclcpp/rclcpp.hpp"
#include "tutorial_interfaces/srv/add_three_ints.hpp"
// CHANGE

#include <chrono>
#include <cstdlib>
#include <memory>

using namespace std::chrono_literals;

int main(int argc, char **argv)
{
  rclcpp::init(argc, argv);

  if (argc != 4) { // CHANGE
      RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "usage: add_three_ints_client X Y Z");
// CHANGE
      return 1;
  }

  std::shared_ptr<rclcpp::Node> node = rclcpp::Node::make_shared("add_three_ints_client");
// CHANGE
  rclcpp::Client<tutorial_interfaces::srv::AddThreeInts>::SharedPtr client =
// CHANGE
    node->create_client<tutorial_interfaces::srv::AddThreeInts>("add_three_ints");
// CHANGE

  auto request = std::make_shared<tutorial_interfaces::srv::AddThreeInts::Request>();
// CHANGE
  request->a = atoll(argv[1]);
  request->b = atoll(argv[2]);
  request->c = atoll(argv[3]);
// CHANGE

  while (!client->wait_for_service(1s)) {
    if (!rclcpp::ok()) {
      RCLCPP_ERROR(rclcpp::get_logger("rclcpp"), "Interrupted while waiting for the service.
Exiting.");
      return 0;
    }
    RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "service not available, waiting again...");
  }

  auto result = client->async_send_request(request);
  // Wait for the result.
  if (rclcpp::spin_until_future_complete(node, result) ==
    rclcpp::FutureReturnCode::SUCCESS)
  {
    RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "Sum: %ld", result.get()->sum);
  } else {
    RCLCPP_ERROR(rclcpp::get_logger("rclcpp"), "Failed to call service add_three_ints");
// CHANGE
  }

  rclcpp::shutdown();
```

```
  return 0;
}
```

CMakeLists.txt

Add the following lines (C++ only):

```
#...

find_package(ament_cmake REQUIRED)
find_package(rclcpp REQUIRED)
find_package(tutorial_interfaces REQUIRED)          # CHANGE

add_executable(server src/add_two_ints_server.cpp)
ament_target_dependencies(server
  rclcpp tutorial_interfaces)                       # CHANGE

add_executable(client src/add_two_ints_client.cpp)
ament_target_dependencies(client
  rclcpp tutorial_interfaces)                       # CHANGE

install(TARGETS
  server
  client
  DESTINATION lib/${PROJECT_NAME})

ament_package()
```

package.xml

Add the following line:

C++ | Python

```
<depend>tutorial_interfaces</depend>
```

After making the above edits and saving all the changes, build the package:

C++ | Python

On Linux/macOS:

```
colcon build --packages-select cpp_srvcli
```

On Windows:

```
colcon build --merge-install --packages-select cpp_srvcli
```

Then open two new terminals, source `ros2_ws` in each, and run:

C++ | Python

```
ros2 run cpp_srvcli server
```

```
ros2 run cpp_srvcli client 2 3 1
```

Summary

In this tutorial, you learned how to create custom interfaces in their own package and how to utilize those interfaces in other packages.

This tutorial only scratches the surface about defining custom interfaces. You can learn more about it in About ROS 2 interfaces.

Next steps

The next tutorial covers more ways to use interfaces in ROS 2.