# Composing multiple nodes in a single process

**Table of Contents**



**Goal:** Compose multiple nodes into a single process.

**Tutorial level:** Intermediate

**Time:** 20 minutes

## Background

See the conceptual article.

For information on how to write a composable node, check out this tutorial.

# Run the demos

The demos use executables from rclcpp_components, ros2component, and composition packages, and can be run with the following commands.

## Discover available components

To see what components are registered and available in the workspace, execute the following in a shell:

```
ros2 component types
```

The terminal will return the list of all available components:

```
(... components of other packages here)
composition
  composition::Talker
  composition::Listener
  composition::NodeLikeListener
  composition::Server
  composition::Client
(... components of other packages here)
```

## Run-time composition using ROS services with a publisher and subscriber

In the first shell, start the component container:

```
ros2 run rclcpp_components component_container
```

Open the second shell and verify that the container is running via `ros2` command line tools:

```
ros2 component list
```

You should see a name of the component:

```
/ComponentManager
```

In the second shell load the talker component (see talker source code):

```
ros2 component load /ComponentManager composition composition::Talker
```

The command will return the unique ID of the loaded component as well as the node name:

```
Loaded component 1 into '/ComponentManager' container node as '/talker'
```

Now the first shell should show a message that the component was loaded as well as repeated message for publishing a message.

Run another command in the second shell to load the listener component (see listener source code):

```
ros2 component load /ComponentManager composition composition::Listener
```

Terminal will return:

```
Loaded component 2 into '/ComponentManager' container node as '/listener'
```

The `ros2` command line utility can now be used to inspect the state of the container:

```
ros2 component list
```

You will see the following result:

```
/ComponentManager
   1  /talker
   2  /listener
```

Now the first shell should show repeated output for each received message.

## Run-time composition using ROS services with a server and client

The example with a server and a client is very similar.

In the first shell:

```
ros2 run rclcpp_components component_container
```

In the second shell (see server and client source code):

```
ros2 component load /ComponentManager composition composition::Server
ros2 component load /ComponentManager composition composition::Client
```

In this case the client sends a request to the server, the server processes the request and replies with a response, and the client prints the received response.

## Compile-time composition with hardcoded nodes

This demo shows that the same shared libraries can be reused to compile a single executable running multiple components without using ROS interfaces. The executable contains all four components from above: talker and listener as well as server and client, which is hardcoded in the main function.

In the shell call (see source code):

```
ros2 run composition manual_composition
```

This should show repeated messages from both pairs, the talker and the listener as well as the server and the client.

> **Note**
>
> Manually-composed components will not be reflected in the `ros2 component list` command line tool output.

### Run-time composition using dlopen

This demo presents an alternative to run-time composition by creating a generic container process and explicitly passing the libraries to load without using ROS interfaces. The process will open each library and create one instance of each “rclcpp::Node” class in the library (source code).

Linux | macOS | Windows

```
ros2 run composition dlopen_composition `ros2 pkg prefix
composition`/lib/libtalker_component.so `ros2 pkg prefix
composition`/lib/liblistener_component.so
```

Now the shell should show repeated output for each sent and received message.

Note

dlopen-composed components will not be reflected in the `ros2 component list` command line tool output.

### Composition using launch actions

While the command line tools are useful for debugging and diagnosing component configurations, it is frequently more convenient to start a set of components at the same time. To automate this action, we can use a launch file:

```
ros2 launch composition composition_demo.launch.py
```

## Advanced Topics

Now that we have seen the basic operation of components, we can discuss a few more advanced topics.

### Unloading components

In the first shell, start the component container:

```
ros2 run rclcpp_components component_container
```

Verify that the container is running via `ros2` command line tools:

```
ros2 component list
```

You should see a name of the component:

```
/ComponentManager
```

In the second shell load both the talker and listener as we have before:

```
ros2 component load /ComponentManager composition composition::Talker
ros2 component load /ComponentManager composition composition::Listener
```

Use the unique ID to unload the node from the component container.

```
ros2 component unload /ComponentManager 1 2
```

The terminal should return:

```
Unloaded component 1 from '/ComponentManager' container
Unloaded component 2 from '/ComponentManager' container
```

In the first shell, verify that the repeated messages from talker and listener have stopped.

## Remapping container name and namespace

The component manager name and namespace can be remapped via standard command line arguments:

```
ros2 run rclcpp_components component_container --ros-args -r __node:=MyContainer -r __ns:=/ns
```

In a second shell, components can be loaded by using the updated container name:

```
ros2 component load /ns/MyContainer composition composition::Listener
```

Note

Namespace remappings of the container do not affect loaded components.

## Remap component names and namespaces

Component names and namespaces may be adjusted via arguments to the load command.

In the first shell, start the component container:

```
ros2 run rclcpp_components component_container
```

Some examples of how to remap names and namespaces.

Remap node name:

```
ros2 component load /ComponentManager composition composition::Talker --node-name talker2
```

Remap namespace:

```
ros2 component load /ComponentManager composition composition::Talker --node-namespace /ns
```

Remap both:

```
ros2 component load /ComponentManager composition composition::Talker --node-name talker3 --node-
namespace /ns2
```

Now use `ros2` command line utility:

```
ros2 component list
```

In the console you should see corresponding entries:

```
/ComponentManager
   1  /talker2
   2  /ns/talker
   3  /ns2/talker3
```

**Note**

Namespace remappings of the container do not affect loaded components.

### Passing parameter values into components

The `ros2 component load` command-line supports passing arbitrary parameters to the node as it is constructed. This functionality can be used as follows:

```
ros2 component load /ComponentManager image_tools image_tools::Cam2Image -p burger_mode:=true
```

### Passing additional arguments into components

The `ros2 component load` command-line supports passing particular options to the component manager for use when constructing the node. As of now, the only command-line option that is supported is to instantiate a node using intra-process communication. This functionality can be used as follows:

```
ros2 component load /ComponentManager composition composition::Talker -e
use_intra_process_comms:=true
```

## Composable nodes as shared libraries

If you want to export a composable node as a shared library from a package and use that node in another package that does link-time composition, add code to the CMake file which imports the actual targets in downstream packages.

Then install the generated file and export the generated file.

A practical example can be seen here: ROS Discourse - Ament best practice for sharing libraries

# Composing Non-Node Derived Components

In ROS 2, components allow for more efficient use of system resources and provide a powerful feature that enables you to create reusable functionality that is not tied to a specific node.

One advantage of using components is that they allow you to create non-node derived functionality as standalone executables or shared libraries that can be loaded into the ROS system as needed.

To create a component that is not derived from a node, follow these guidelines:

1. Implement a constructor that takes `const rclcpp::NodeOptions&` as its argument.
2. Implement the `get_node_base_interface()` method, which should return a `NodeBaseInterface::SharedPtr`. You can use the `get_node_base_interface()` method of a node that you create in your constructor to provide this interface.

Here's an example of a component that is not derived from a node, which listens to a ROS topic: node_like_listener_component.

For more information on this topic, you can refer to this discussion.