Snapcraft

Distributing ROS apps with snaps

Developer Guide - Part 1



Distribute robotics applications like a global software vendor

Robotics developers know app development inside-out, but deploying a robotics application can be challenging. It's not uncommon to compile the code on robots, copy/paste compiled packages and end up with unknown versions of software. Even worse, one can experience the infamous "It works on my machine" syndrome.

Robotics software should benefit from a controlled and stable environment, with the same portability and reliability as any other software application. Achieving this reliability should be simple, even if our software relies on hundreds of dependencies.

What we will learn

This developer guide will take us through the process of packaging our first ROS application as a snap.

Snaps offer a solution to build and distribute containerised robotics applications or any software. It is the de facto distribution tool for companies deploying software on Ubuntu, like Microsoft, Google, Spotify and more. As such, we will be able to leverage the same tooling and global infrastructure for our application.

Across this developer guide, we will explore how to build snaps for a robotics application. Through different examples, we will cover the basics of snap creation for a ROS and ROS 2 application. By introducing the main concepts behind snaps, we will see how to confine our robotics application and make it installable on dozens of Linux distributions.

Requirements

We will need an up and running Ubuntu (or similar) operating system. The version doesn't matter as long as it's at least 16.04. The installation could be native or in a VM. If using a container, we must be sure that we can install and run systemd, snapd and snaps.

In addition, we will need basic knowledge about ROS or ROS 2 as well as some basic understanding of the Linux environment (Ubuntu).

This developer guide has been tailored for robotics developers looking for a solution to deploy their robotics software and applications. No previous experience with snaps is necessary.



What is a snap?

Snaps are the perfect solution for software deployment in embedded Linux devices.

Snaps are containers that bundle an application and all its dependencies, offering roboticists:

- **A containerised solution**: snaps bundle all our dependencies and assets in one package, making our application installable on dozens of Linux distributions and across distro versions. We won't even have to install anything else on our robots' operating system, no dependencies, not even ROS if we are using it.
- **Strict confinement**: snaps are designed to be secure and isolated from the underlying system and other applications, with dedicated interfaces to access the host machine.
- **CI/CD integration**: the creation of snaps can be integrated into our CI pipeline, making the updates effortless.
- **OTA and delta updates**: snaps can update automatically and transactionally, making sure the device is never broken.
- **Multi-architecture**: snaps come with a multi-architecture feature, allowing us to build our snap package for multiple architectures.
- **Managing updates**: snaps can be updated automatically, or we can control the update options (update hours, update holds, update history). It also comes with multiple release channels for role-based access controls and application versioning.
- **Reduce boot time**: We can configure a snap application as a daemon, so it starts automatically at boot.

What can snaps do for our robotics applications?

Snaps are meant to deploy software that has been developed and tested.

Snaps offer a solution to deploy and distribute our software. It's an alternative package manager (like APT). With snaps, we can manage updates and keep track of the version installed on our robot without ever breaking our installation. Sharing and deploying our application to all our users or all our devices is made easy. With snaps, we can seamlessly run an application on our distro and access the host machine and its resources securely.

As such, a snap is a solution to deploy our robotics applications.

What can't snaps do for our robotics applications?

Snaps are not meant for testing and debugging.

Snaps don't embed our source code. As such, developers can't use snaps to test some fresh code. It's not a distribution mechanism for early debugging sessions. On its own, snaps are not meant for cloud web services deployments or web

services applications; i.e. Docker. Snaps are used in environments where secure access to the host machine and resources is crucial.

Snaps were designed for embedded Linux applications, with optimisations for ROS packaging.

Install snapd, snapcraft and LXD

Snapd

`Snapd` is a daemon required to download, install and run snaps. `Snapd` also includes the snap command, used to communicate with `snapd`.

Installing `snapd` is straightforward in most Linux distributions. `Snapd` comes pre-installed on most Ubuntu flavours, starting from 16.04.
In most cases, `snapd` can be installed with:

```
sudo apt update
sudo apt install snapd
```

For different distributions, we can refer to the documentation to install snapd.

`Snapd` is also available through a snap. We can get a more recent version of `snapd` with:

```
sudo snap install snapd
```

After installing snapd we should be able to type in a terminal:

```
snap --version
```

A quick look at `snap help` will show everything that we can do with the snap command:

```
$ snap help

Commonly used commands can be classified as follows:

Basics:          find, info, install, remove, list
...more:         refresh, revert, switch, disable, enable, create-cohort
History:         changes, tasks, abort, watch
Daemons:         services, start, stop, restart, logs
Permissions:     connections, interface, connect, disconnect
Configuration:   get, set, unset, wait
AppAliases:      alias, aliases, unalias, prefer
Account:         login, logout, whoami
Snapshots:       saved, save, check-snapshot, restore, forget
Device:          model, reboot, recovery
...Other:        warnings, okay, known, ack, version
Development:     download, pack, run, try
```

Snapcraft

While `snapd` is used to download and run snaps, snapcraft is the tool we need to build snaps.

If we installed `snapcraft` from Debian previously, we will have to uninstall it, the Debian package is no longer updated. To do so, just run: `sudo apt remove snapcraft --purge`

To install snapcraft simply run:

```
sudo snap install snapcraft --classic
```

The `--classic` flag refers to the confinement. We will address the confinement topic later in this guide.

`Snapcraft` is not only a tool to build snaps, but more generally a developer tool. Meaning that we will need it to build, upload and share our snaps.

LXD

LXD *is the container technology used by snaps to isolate our snap build. `LXD` is not only dedicated to snaps. It's a next generation system container and virtual machine manager.*

LXD can be installed with a snap and must be configured to be used. To install `LXD` simply run:

```
sudo snap install lxd
```

Now that `LXD` is installed, we must configure it. We are going to use a default profile to do so:

```
sudo lxd init --auto
```

We can make sure everything went well by listing the profiles and making sure the default profile is listed:

```
$ lxc profile list
+------------+----------------------+------------+
|  NAME      |  DESCRIPTION         | USED BY    |
+------------+----------------------+------------+
| default    | Default LXD profile | 0          |
+------------+----------------------+------------+
```

First ROS 2 snap

Our first snap will be a basic ROS 2 Humble talker-listener. We are going to use ros2_demos: demo_nodes_cpp. It contains a talker publishing a message and a listener subscribing to it. Both nodes can be launched with the help of the talker_listener.launch.py.

Understanding the snapcraft.yaml file

First clone the package from GitHub:

```
git clone https://github.com/ubuntu-robotics/ros2-humble-talker-listener-snap.git
```

The repository contains a snap folder with a `snapcraft.yaml file`.

Snaps are defined in a single `YAML` file placed in our project. Let's explore our snapcraft.yaml and break it down in the next section:

Metadata

The `snapcraft.yaml` file starts with a small amount of human-readable metadata, which usually can be lifted from the GitHub description or project README.md. This data is used in the presentation of our app in the Snap Store (for example, see PlotJuggler front page).

```
name: ros2-talker-listener
version:      '0.1'
summary:      ROS 2 Talker/Listener Example
description:  |
  This example launches a ROS 2 talker and listener.
```

The `name` must be unique in the Snap Store. Valid snap names consist of lower-case alphanumeric characters and hyphens. They cannot be all numbers, and they cannot start or end with a hyphen.

This is a declarative `version` of the packaged software and is not linked to the version of the snap software itself. It's also possible to write a script to calculate the version, or to take a tag or commit from a git repository.

The `summary` cannot exceed 79 characters.

- For more information about core versions, please see top-level-metadata.

Base

Next in the `YAML` file we will find the `base` keyword. This defines a special kind of snap that provides a run-time environment with a minimal set of libraries that are common to most applications. They're transparent to users, but they need to be considered, and specified, when building a snap. As snap developers, we should consider it, but our users will be able to install the final snap independently of their OS version.

```
base: core22
```

core22 is the current standard base for snap building and is based on Ubuntu 22.04 LTS. It is therefore the base for ROS 2 Humble snaps.

- For more information about core versions, please refer to the appendix.

Security model

Following, we will see the `confinement` keyword. Snaps are containerised to ensure predictable application behaviour and to provide greater security. We will review

this topic later in this guide.

To get started, we won't confine this application. Unconfined applications are specified with devmode.

```
confinement: devmode
```

- For more information about security models, please refer to the confinement section.

Parts

`Parts` follow next. They define how to build our app and can be anything: programs, libraries, or other assets needed to create and run our application. Their source can be local directories, remote git repositories, or tarballs. Multiple `parts` can be defined within the same `snapcraft.yaml` in order to build dependencies or even an additional application.

In this example, we have a single part: '`ros-demos`'.

```
parts:
  ros-demos:
    plugin: colcon
    source: https://github.com/ros2/demos.git
    source-branch: humble
    source-subdir: demo_nodes_cpp
    stage-packages: [ros-humble-ros2launch]
```

Snapcraft relies on well known and well established ROS tools such as, in this example, colcon. Plugins allow us to identify such tools.

The packages we're building must have `install` rules, or else snapcraft won't know which components to place into the snap. We should make sure we install binaries, libraries, header files, launch files, etc. Here, we selected the Humble branch of ros2-demos github repository as `source-branch`.

Since `ros2-demos` contains multiple packages, we select `demo_nodes_cpp` with the `source-subdir` entry.

We notice that `ros-humble-ros2launch` is listed as a `stage-packages`. Stage packages are packages required to run the `part`. Usually this exec dependency is missing from the `package.xml` hence we must specify it. The rest of the dependencies are going to be automatically downloaded with rosdep based on the `package.xml`.

- For more information about general parts metadata, see parts-metadata.
- For more information about plugins, please refer to the appendix.

Apps

After `Parts` we find the `Apps` keyword. These are the commands and services exposed to end users.

```
apps:
  ros2-talker-listener:
```

```
    command: opt/ros/humble/bin/ros2 launch demo_nodes_cpp talker_listener.
launch.py
    extensions: [ros2-humble]
```

The entry under `apps` is the app name that should be exposed to the end users. In our case the app name is `ros2-talker-listener`.

In snap, an application is usually prefixed by the snap name so that the application `my-app` from the snap `my-snap` can be executed by calling `my-snap.my-app`. However, if both the snap and the app are called the same, as is the case in our ROS 2 example, the execution command collapses to avoid the tediousness of writing twice the same words.

As a result, the command `ros2-talker-listener.ros2-talker-listener` simply becomes `ros2-talker-listener`. We will see this when we run the snap.

Multiple `apps` can be defined within the same `snapcraft.yaml`. Our snap will then expose multiple commands.

Then, after the app name, we find the `command` entry. This specifies the path to the binary to be run, along with arguments. This is resolved relative to the root of our snap contents (hence there is no '`/`' before `opt`).

Finally, the ros2-humble extension will set our ROS 2 humble build and runtime environment. This way we don't have to source manually our ROS 2 environment.

- For more information about ROS extensions, please refer to the appendix.

Building, installing and running a snap

Now that our `snapcraft.yaml` is ready, we will describe how to build our package. In this section, we will also cover how to install and run the created snap.

Building

The file `snapcraft.yaml` is expected to be found in the `snap/` directory. So the `snapcraft` command is expected to be run at the root of our repository where we can find the `snap/` directory. The snapcraft command is going to look for `snap/snapcraft.yaml` and start building our snap.

Snapcraft is building the snap in steps:

1. **pull**: downloads or otherwise retrieves the components needed to build the part.

2. **build**: constructs the part from the previously pulled components. The plugin of a part specifies how it is constructed.

3. **stage**: copies the built components into the staging area. This is the first time all the different parts that make up the snap are actually placed in the same directory.

4. **prime**: copies the staged components into the priming area, to their final locations for the resulting snap. This is very similar to the stage step, but files

go into the priming area instead of the staging area. The prime step exists because the staging area might still contain files that are required for the build but not for the snap.

To build our snap, we will run:

```
snapcraft
```

This will take some time, but once it's done we will see:

```
Created snap package ros2-talker-listener_0.1_amd64.snap
```

This `.snap` file is our packaged application.

Installing

If we could install `snapd` on our current distribution, it means that we can install our freshly built snap. We don't need to be running Ubuntu 22.04 to run this ROS 2 Humble snap. We don't even need to install ROS on our host system to install and run the snap.

Snaps bundle all their dependencies as well as their "`core`" which make them host-agnostic.

Since our snap is currently not confined, we will install it with the flag `--devmode`.

```
sudo snap install ros2-talker-listener_0.1_amd64.snap --devmode
```

Now our snap is installed on our system!

Running the snap

Now let's run the snap that we just installed.

We can start the snap by running:

```
ros2-talker-listener
```

We will see the talker listener starting to exchange messages. We can then `ctrl-c` it to stop it.
Note that we built, installed and ran this ROS application without even installing ROS 2 on our host.

We can verify that by running:

```
ls -la /opt
```

We can see that nothing is listed in the installation folder.

In this example, our snap has only one app, but snap can contain as many applications (commands) as we need. We can easily get info on our installed snap with the `snap info` command:

```
$ snap info ros2-talker-listener

name:         ros2-talker-listener
summary:      ROS 2 Talker/Listener Example
publisher:    -
license:      unset
description:  |
  This example launches a ROS 2 talker and listener.
Commands:
  - ros2-talker-listener
refresh-date:  today at 10:54 CEST
installed:    0.1 (x1) 64MB devmode
```

We can see all kinds of metadata as well as the commands available from the snap.

Confining our first snap application

Our application was installed in `devmode`. This means that our snap can access every resource from our host system (files, devices, etc.). For security, snaps are meant to be run and distributed as strictly confined applications.

In this section we will explore the confinement types, grades and interfaces available. Then, we will strictly confine our application.

Confinement types

So far, in our `snapcraft.yaml`, we only declared:

```
confinement: devmode
```

Let's have a closer look at the types of confinement:

Devmode

A special mode for snap creators and developers. A *devmode* snap runs as a strictly confined snap with full access to system resources, and produces debug output to identify unspecified interfaces. Installation requires the --devmode command line argument.

Classic

Allows access to our system's resources in much the same way traditional packages do. To safeguard against abuse, publishing a classic snap requires manual approval, and installation requires the --classic command line argument. The typical applications allowed with classic confinement are IDEs (vscode, qtcreator).

Strict

Used by the majority of snaps. Strictly confined snaps run in complete isolation, up to a minimal access level that's deemed always safe. Consequently, strictly

confined snaps can not access our files, network, processes or any other system resource without requesting specific access via an interface.

In this case, our application should be confined as strict since we want to be able to share it securely. Everything it needs to access can be declared through interfaces. Let's make that change:

```
confinement: strict
```

- For more information about security models, please see choosing security models.

Grade

By adding the `grade` keyword, we can declare the quality of our snap. By defining the grade, we can make sure that a development version never goes into a stable channel.

There are only two grades possible:

Devel

A `devel` snap indicates that this is a development version, and it is not meant to be released as stable.

Stable

The default one. Meant for production grade snaps, so it can later be released to every user.
For this example, let's add the `grade` keyword and select `stable`:

```
grade: stable
```

Interfaces

Interfaces enable resources from one snap to be shared with another or with the system. An interface consists of a connection between a slot and a plug. The slot is the provider of the interface while the plug is the consumer, and a slot can support multiple plug connections.

The list of available interfaces is available on the online documentation. We can find interfaces to access the home directory, the CANBus, the network etc.

The interfaces to use are declared in the `snapcraft.yaml` for each application. In the online documentation, we will see that some interfaces are listed as "`auto-connect=yes`" and some are not. The auto-connectable interface will connect at the installation of the snap, while the others will have to be connected manually. This resembles the security validation of an app requesting permissions to the user to access some resources.

Which interfaces a snap requires, and provides, is very much dependent on the type of snap and its own requirements.

For our ROS 2 snap, we will need two auto-connect interfaces: `network` to enable network access and `network-bind` to let our snap operate as a network service.

```
plugs: [network, network-bind]
```

- For more information about interfaces, please see the online documentation.

Confining and rebuilding our snap

We will confine our application by changing the confinement level and adding the grade and plugs in our `snapcraft.yaml`.

We can find the updated code from the "confined" branch.

To switch to the confined branch:

```
git switch confined
```

Our `snapcraft.yaml` file should have these modifications:

```
-confinement: devmode
+confinement: strict
+grade: stable
 apps:
  ros2-talker-listener:
   command: opt/ros/humble/bin/ros2 launch talker-listener talker_listener.
launch.py
+   plugs: [network, network-bind]
```

Once our changes are done, let's rebuild our snap:

```
snapcraft
```

This time our snap is confined, so we don't need the `--devmode` flag any more. Yet, we will need the `--dangerous` flag, since our snap hasn't been signed by an official store.

To install our snap:

```
sudo snap install ros2-talker-listener_0.1_amd64.snap --dangerous
```

Let's check the connections of our freshly confined snap. Run:

```
$ snap connections ros2-talker-listener

Interface       Plug                                Slot          Notes
network         ros2-talker-listener:network        :network      -
network-bind    ros2-talker-listener:network-bind   :network-bind -
```

In the above output, we can see that our snap is connected to the network and network-bind slot.

If everything looks good, let's run our confined snap:

```
ros2-talker-listener
```

We will now face this log:

```
[talker-1] 2022-07-13 15:47:10.570 [RTPS_TRANSPORT_SHM Error] Failed to create segment cbbe40933e75c60a: Permission denied -> Function compute_per_allocation_extra_size

[listener-2] 2022-07-13 15:47:10.654 [RTPS_TRANSPORT_SHM Error] Failed to create segment 59f8e836a0800439: Permission denied -> Function compute_per_allocation_extra_size

[talker-1] 2022-07-13 15:47:10.657 [RTPS_MSG_OUT Error] Permission denied -> Function init

[listener-2] 2022-07-13 15:47:10.657 [RTPS_MSG_OUT Error] Permission denied -> Function init

[talker-1] [INFO] [1657720031.730359135] [talker]: Publishing: 'Hello World: 1'

[listener-2] [INFO] [1657720031.730705569] [listener]: I heard: [Hello World: 1]

[talker-1] [INFO] [1657720032.730352803] [talker]: Publishing: 'Hello World: 2'

[listener-2] [INFO] [1657720032.730571444] [listener]: I heard: [Hello World: 2]
```

The error that we see is related to the shared memory transport not being able to create its file. This is expected. We will cover how to manage shared memory in our snaps later. For now, even with the shared memory failing, ROS 2 falls back to network transport (UDP) and messages are properly sent and received.

Apart from our shared memory error message, our snap is now running strictly confined with only access to our network.

Run our snap as a daemon

One of the advantages of using snaps is that they can turn our application into a service (or a daemon) in an incredibly easy way. Once we have turned our application into a service, it can automatically start at boot and end when the machine is shut down. We can also start and stop on demand through socket activation.

A daemon can take different forms, where the first two daemons are the most used forms:

- **simple:** Run for as long as the service is active - this is typically the default option.
- **oneshot:** Run once and exit after completion, notifying systemd.
- **forking:** The configured command calls fork() as part of its start-up, and the parent process is then expected to exit when start-up is complete.
- **notify:** Assumes the command will send a signal to systemd to indicate its running state.

The daemon feature is per command and not per snap. This means that one snap

containing multiple commands can have some running as daemon and others running as simple commands.

To turn our talker-publisher into a simple daemon, we will add `daemon: simple` in our `snapcraft.yaml`.

```
 apps:
  ros2-talker-listener:
    command: opt/ros/humble/bin/ros2 launch talker-listener talker_listener.
launch.py
+   daemon: simple
    plugs: [network, network-bind]
    extensions: [ros2-humble]
```

That's it.
To switch to this version of the file run:

```
git switch daemon
```

Now let's rebuild and install our snap. The build should be quicker as this has been done before.

```
snapcraft
sudo snap install ros2-talker-listener_0.1_amd64.snap --dangerous
```

Let's explore what happens now that we have turned our snap application into a daemon.
First, we get some info about our snap:

```
$ snap info ros2-talker-listener

name:             ros2-talker-listener
summary:          ROS 2 Talker/Listener Example
publisher:        –
license:          unset
description:      |
  This example launches a ROS 2 talker and listener.
services:
  ros2-talker-listener: simple, enabled, active
refresh-date:     today at 16:39 CEST
installed:        0.1 (x9) 64MB -
```

Our command is now listed as a service and marked enabled and active. This means that our `talker-listener` is currently running as a daemon.

`Active` means the talker-listener is now running in the background. And `enabled` means that it will automatically start at boot and restart in case of failure.

Log our service

We can inspect the log of our running service with the `snap` tool. To inspect our snap log run:

```
$ sudo snap logs ros2-talker-listener

ros2-talker-listener.ros2-talker-listener[970635]: [talker-1] [INFO]
[1657724117.996643358] [talker]: Publishing: 'Hello World: 957'
...
ros2-talker-listener.ros2-talker-listener[970635]: [listener-2] [INFO]
[1657724121.996731460] [listener]: I heard: [Hello World: 961]
```

We can also add the `-f` flag if we want to wait for new lines and print them as they come in.

Snap logs are actually available in the systemd journal. Hence, we can log a snap service directly from the `journalctl` command:

```
journalctl -fu snap.ros2-talker-listener.ros2-talker-listener.service
```

(`-f` for follow and `-u` to show the log of our specific unit)

Interact with our service

We saw that our service was `enabled` and `active`, we can obviously interact with these states.

In case we want to temporarily stop our service, we can by using running:

```
sudo snap stop ros2-talker-listener
```

We have stopped our service, so it's no longer running in the background. But if we reboot, our service will still start automatically. This is because our service is still `enabled`.

To disable our service, we can do it with:

```
sudo snap --disable ros2-talker-listener
```

Now our service won't start again. We can verify the result of our actions by running the `snap info` command on our snap.

We have seen how to `stop/disable` our service, but of course we also have the corresponding `start/enable` command. Please visit the documentation to know more about service-management.

Exercise

Let's do a little exercise. Right now, our talker-listener only has one application. How about you add another one? Having the ROS 2 topic tools inside our snap would be great for some introspection.

Assignment

Modify the `snapcraft.yaml` file:

- Add a new app entry for ros2topic
- Add the corresponding dependencies

Build and install the new snap.

Outcome

By the end of this exercise you will have a new command available:

```
ros2-talker-listener.ros2topic hz /chatter
```

- Note that the ROS 2 topic tools are only going to work with message types that are included in the snap.

You can review the solution to this exercise here.

Conclusion

In this developer guide, we went through the creation of a basic ROS 2 snap. But in the process, we learned the basics to create our own ROS snap as well. We have covered the basic concepts of a snap, how to build and run them. We also shared the benefits and good development practices. There are more features and advanced development tips that we have yet to cover.

This guide is the first part of a series dedicated to robotics deployment. The second part will focus on building and running Turtlebot 3 software in a strictly confined snap. We will also cover the Snap Store and how we can use it to deploy our snap to production and its management. Finally, the third part of the series will cover Ubuntu Core; a secure, application-centric OS for embedded devices.

Visit the snapcraft robotics documentation to go further. If we have any questions or need help, we can visit and post our question on the snapcraft forum.

For more information about Ubuntu and robotics please visit our website.

You may also consider reading the following materials:

- ROS snapcraft documentation
- Docker & ROS: When all you have is a hammer, everything looks like a nail
- ROS 2 shared memory and snaps
- Key considerations when choosing a robot's operating system

Need help in getting to market? contact us

Appendix

Plugins

Snapcraft plugins are used to build our parts in the `snapcraft.yaml` file. This is the plugin which is going to pull our dependencies, build and install our software in the snap. There are plugins available for various programming languages and build tools. `Colcon` is not the only ROS specific plugin that is available with snaps. Here is the exhaustive list:

catkin

The catkin plugin is useful when building ROS 1 parts. It comes available for ROS Melodic and Noetic. The plugins enable plugin-specific keywords in addition to the generic part's keyword.

catkin-tools

Similarly, the catkin-tools plugin runs the configuration process that allows catkin_tools to run. It's also available for ROS Melodic and Noetic and comes with plugin-specific features and syntax.

colcon

In the parts section we already presented the colcon plugin. This plugin is also available for ROS 2 Dashing and Foxy.

We can get detailed information about the plugin options with the command:

```
snapcraft help **plugin_name**
```

ROS extensions

ROS extensions help us snap ROS 1 and ROS 2 applications. They come with the respective plugins, add the APT package repositories and set a build and run-time environment. Extensions are saving us from writing a lot of boilerplate code in our `snapcraft.yaml`. They should be declared in the part section of our `snapcraft.yaml` file.

There are three other extensions available to help us build and run a ROS snap:

ros-noetic

The ros-noetic extension helps us snap ROS Noetic applications. It comes along with the caktin or catkin-tools plugins. The extensions add the ROS APT package repository and set a ROS noetic build and run-time environment.

ros2-foxy

The ros2-foxy extension helps us snap ROS 2 Foxy applications. It comes along with the `colcon` plugin. The extensions add the ROS 2 APT package repository and set a ROS 2 Foxy build and run-time environment.

ros2-humble

The ros2-humble extension helps us snap ROS 2 Humble applications. It comes along with the `colcon` plugin. The extensions add the ROS 2 APT package repository and set a ROS 2 Humble build and run-time environment.

Extensions automatically fill up some fields which are common to the corresponding ROS-release snaps, sparing us some typing. Note that we can 'expand' the extension and reveal all of its secrets by issuing the following command:

```
snapcraft expand-extensions
```

Notice that there are no extensions for ROS Melodic and ROS 2 Dashing. This is due to a design change from `core20`. Before that, all the build-time/run-time of ROS environment was included in the `colcon` or `catkin` plugin.

In this guide we explained in detail the ROS 2 Humble `core22` case. Similarly, we can use the extensions for ROS melodic/noetic or ROS 2 dashing/foxy. But even if an extension is not available for our current ROS distribution, we can still snap it. For more information, visit the next link.

Core versions

Core versions are base snaps that should be declared in the `snapcraft.yaml` file using the base keyword. They provide a run-time environment with a minimal set of libraries that are common to most applications. They're transparent to users, but they need to be considered and specified when building a snap.

Core versions are sharing the same base as the corresponding Ubuntu LTS. Meaning that core22 corresponds to Ubuntu 22.04, etc.

`Core` version doesn't have to match the host, but has to match the ROS version that we want to install and run from our snap. This means that once built, we can install our ROS 2 Humble snap on dozens of Linux distributions (for example, an Ubuntu 18.04 or the latest Manjaro release).

While we have used core22 base in this guide, there are different core versions available:

Core18

The core version for ROS Melodic and ROS 2 Dashing.

Core20

The core version for ROS Noetic and ROS 2 Foxy.

Core22

The core version for ROS 2 Humble.

Solution to the exercise

Here is a solution for the proposed exercise.

- Add `ros2topic` as a `stage-package` to have it available at run time:

```
 parts:
  ros-demos:
    plugin: colcon
    source: https://github.com/ros2/demos.git
    source-branch: humble
    source-subdir: demo_nodes_cpp
-   stage-packages: [ros-humble-ros2launch]
+   stage-packages: [ros-humble-ros2launch, ros-humble-ros2topic]
```

We add our dependency (ros-humble-ros2topic) to the stage-packages list. Stage-packages are packages that are required to run the part. Here we decided to add our dependency to our already existing ros-demos part for simplicity. Alternatively, we could have created an additional empty part simply to add our stage-packages dependency.

- Add an additional app:

```
apps:
  ros2-talker-listener:
    command: opt/ros/humble/bin/ros2 launch talker-listener talker_listener.
launch.py
    daemon: simple
    plugs: [network, network-bind]
    extensions: [ros2-humble]
+ ros2topic:
+   command: opt/ros/humble/bin/ros2 topic
+   plugs: [network, network-bind]
+   extensions: [ros2-humble]
```

We added another application entry to our snap. We kept the plugs and extensions and simply exposed the ros2 topic command.

- Build and install the new snap:

```
snapcraft
sudo snap install ros2-talker-listener_0.1_amd64.snap --dangerous
```

- Make sure everything works:

```
ros2-talker-listener.ros2topic hz /chatter
average rate: 1.000 min: 0.999s max: 1.000s std dev: 0.00049s window: 3
```

© Canonical Limited 2023. Ubuntu, Kubuntu, Canonical and their associated logos are the registered trademarks of Canonical Ltd. All other trademarks are the properties of their respective owners. Any information referred to in this document may change without notice and Canonical will not be held responsible for any such changes.

Canonical Limited, Registered in Isle of Man, Company number 110334C, Registered Office: 2nd Floor, Clarendon House, Victoria Street, Douglas IM1 2LN, Isle of Man, VAT Registration: GB 003 2322 47

