

Note: This tutorial assumes that you have completed the previous tutorials: examining the simple publisher and subscriber (/ROS/Tutorials/ExaminingPublisherSubscriber), [Snapcraft tour](#) (<https://snapcraft.io/create/>).

💡 Please ask about problems and questions regarding this tutorial on answers.ros.org (<http://answers.ros.org>). Don't forget to include in your question the link to this page, the versions of your OS & ROS, and also add appropriate tags.

Packaging your ROS project as a snap

Description: This tutorial covers how to package and deploy your ROS project as a snap.

Tutorial Level: INTERMEDIATE

Next Tutorial: Writing a Tutorial (/WritingTutorials)

Sommaire

1. What are snaps?
2. Creating a snap
 1. The snapcraft.yaml
 2. The snapcraft.yaml explained
 3. Actually create the snap
3. Testing the snap
4. Sharing the snap with the world

1. What are snaps?

[Snaps](#) (<https://snapcraft.io/>) are packages that bundle an application and its dependencies. They offer a number of features that address important concerns as one gets closer to shipping a robotic platform:

- Since dependencies are bundled, all dependencies (including ROS) are frozen and won't break from underneath you until you can test them (and release an update)
- They're designed to be secure and isolated from the underlying system and other applications
- They update automatically and transactionally, making sure the device is never broken (you can use the free Ubuntu store, no need to host your own infrastructure)
- They support multiple release channels, making A/B testing easy

2. Creating a snap

This tutorial will demonstrate how to use [Snapcraft](#) (<https://github.com/snapcore/snapcraft>) to create a new snap, and what you can do with it after it has been created.

First, install Snapcraft. You can install it from source (read [HACKING.md](#) (<https://github.com/snapcore/snapcraft/blob/master/HACKING.md>), or if you're using Ubuntu (or an Ubuntu-based distro),

install it from the Snap Store:

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

(You can use the apt repositories if you like, although that version of snapcraft can be a tad old.)

Snapcraft has built-in support for Catkin: you point it at a workspace, and tell it what packages to include in the snap. In order for it to know which project components to include, you must ensure that your projects have good install rules. Let's do that now. Open up your CMakeLists.txt (/catkin/CMakeLists.txt) with `roscd beginner_tutorials CMakeLists.txt`. If you followed the Python (/ROS/Tutorials/WritingPublisherSubscriber%28python%29) tutorial, add the install target at the end so that it looks like this (unused bits and comments removed):

```
cmake_minimum_required(VERSION 2.8.3)
project(beginner_tutorials)

## Find catkin and any catkin packages
find_package(catkin REQUIRED COMPONENTS roscpp rospy std_msgs genmsg)

## Declare ROS messages and services
add_message_files(FILES Num.msg)
add_service_files(FILES AddTwoInts.srv)

## Generate added messages and services
generate_messages(DEPENDENCIES std_msgs)

## Declare a catkin package
catkin_package()

include_directories(include ${catkin_INCLUDE_DIRS})

## Install scripts
install(PROGRAMS scripts/talker.py scripts/listener.py
  DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION}
)
```

If you followed the C++ (/ROS/Tutorials/WritingPublisherSubscriber%28c%2B%2B%29) tutorial, add the install target at the end so that the CMakeLists.txt (/catkin/CMakeLists.txt) looks like this:

```
cmake_minimum_required(VERSION 2.8.3)
project(beginner_tutorials)

## Find catkin and any catkin packages
find_package(catkin REQUIRED COMPONENTS roscpp rospy std_msgs genmsg)

## Declare ROS messages and services
add_message_files(FILES Num.msg)
add_service_files(FILES AddTwoInts.srv)

## Generate added messages and services
generate_messages(DEPENDENCIES std_msgs)

## Declare a catkin package
catkin_package()

## Build talker and listener
include_directories(include ${catkin_INCLUDE_DIRS})

add_executable(talker src/talker.cpp)
target_link_libraries(talker ${catkin_LIBRARIES})
add_dependencies(talker beginner_tutorials_generate_messages_cpp)

add_executable(listener src/listener.cpp)
target_link_libraries(listener ${catkin_LIBRARIES})
add_dependencies(listener beginner_tutorials_generate_messages_cpp)

## Install talker and listener
install(TARGETS talker listener
  RUNTIME DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION}
)
```

Change to the root of the catkin workspace you used in the "writing a simple publisher and subscriber" tutorial:

```
# This workspace should contain the beginner_tutorials package.
$ cd ~/catkin_ws
```

Initialize a new Snapcraft project here:

```
$ snapcraft init
Created snap/snapcraft.yaml.
Edit the file to your liking or run `snapcraft` to get started
```

2.1 The snapcraft.yaml

Open that `snap/snapcraft.yaml` file and make it look like this:

```
name: publisher-subscriber
version: '0.1'
summary: ROS Example
description: |
  The ROS publisher/subscriber example packaged as a snap.

grade: stable
confinement: strict

parts:
  workspace:
    plugin: catkin
    rosdistro: kinetic
    catkin-packages: [beginner_tutorials]

apps:
  roscore:
    command: roscore
    plugs: [network, network-bind]

  talker:
    command: rosrun beginner_tutorials talker
    plugs: [network, network-bind]

  listener:
    command: rosrun beginner_tutorials listener
    plugs: [network, network-bind]
```

2.2 The snapcraft.yaml explained

Let's break that down piece by piece.

```
name: publisher-subscriber
version: '0.1'
summary: ROS Example
description: |
  The ROS publisher/subscriber example packaged as a snap.
```

This is the basic metadata that all snaps require. These fields are fairly self-explanatory, but note that the name must be globally unique among all snaps.

```
grade: stable
confinement: strict
```

`grade` can be either `stable` or `devel`. If it's `devel`, the store will prevent you from releasing into one of the two stable channels (stable and candidate, specifically)-- think of it as a safety net to prevent accidental releases. If it's `stable`, you can release it anywhere.

`confinement` can be `strict`, `devmode`, or `classic`. `strict` enforces confinement, whereas `devmode` allows all accesses, even those that would be disallowed under `strict` confinement (and

logs accesses that would otherwise be disallowed for your reference). `classic` is even less confined than `devmode`, in that it doesn't even get private namespaces anymore (among other things). There is [more extensive documentation on confinement available \(https://snapcraft.io/docs/reference/confinement\)](https://snapcraft.io/docs/reference/confinement).

```
parts:
  workspace:
    plugin: catkin
    rosdistro: kinetic
    catkin-packages: [beginner_tutorials]
```

Snapcraft is responsible for taking many disparate parts and orchestrating them all into one cohesive snap. You tell it the parts that make up your snap, and it takes care of the rest. Here, we tell Snapcraft that we have a single part called `workspace`. We specify that it builds with Catkin, and also specify that we're using Kinetic here as opposed to another ROS distribution. Finally, we specify the packages in this workspace that we want included in the snap. In our case, we only have one: the `beginner_tutorials` package we've been working on through the tutorials.

```
apps:
  roscore:
    command: roscore
    plugs: [network, network-bind]

  talker:
    # Use talker.py if you followed the Python tutorial
    command: rosrun beginner_tutorials talker
    plugs: [network, network-bind]

  listener:
    # Use listener.py if you followed the Python tutorial
    command: rosrun beginner_tutorials listener
    plugs: [network, network-bind]
```

Now things get a little interesting. When this snap is built, it will include a complete ROS system: `roscpp`, `rospy`, `roscore`, your ROS workspace, etc. It's a standalone unit, and you're in complete control over how the user interacts with it. You exercise that control via the `apps` keyword, where you expose specific commands to the user.


Here we simply expose the three components of the publisher/subscriber tutorial: `roscore`, the `talker`, and the `listener`. We could just as easily written a [launch file \(http://wiki.ros.org/roslaunch\)](http://wiki.ros.org/roslaunch) to bring up the entire system in a single app.

We use `plugs` to specify that each app requires network access ([read more about interfaces \(https://snapcraft.io/docs/core/interfaces\)](https://snapcraft.io/docs/core/interfaces)). Without this, each app would be confined such that they couldn't communicate.

2.3 Actually create the snap

That's it: time to build the snap.

```
$ cd ~/catkin_ws
$ snapcraft
```

That will take a few minutes. You'll see Snapcraft fetch  rosdep (<http://docs.ros.org/independent/api/rosdep/html/>), which is then used to determine the dependencies of the ROS packages in the workspace. It then pulls those down and puts them into the snap along with roscore. Finally, it builds the requested packages in the workspace, and installs them into the snap as well. At the end, you'll have your snap.

3. Testing the snap

As we discussed previously, this snap is completely standalone: you could email it to someone and they'd be able to install it and run your ROS system, even if they didn't have ROS installed themselves. Test it out yourself:

```
# We use --dangerous here because the snap doesn't come from an
# authenticated source
$ sudo snap install --dangerous publisher-subscriber_0.1_amd64.snap
```

Now you can take it for a spin just like you did when examining the simple publisher and subscriber (/ROS/Tutorials/ExaminingPublisherSubscriber). First, run roscore:

```
$ publisher-subscriber.roscore
```

Now run the publisher:

```
$ publisher-subscriber.talker
```

And you'll see the familiar output:

```
[ INFO] [1497643945.491444894]: hello world 5
[ INFO] [1497643945.591430533]: hello world 6
[ INFO] [1497643945.691426519]: hello world 7
[ INFO] [1497643945.791444793]: hello world 8
[ INFO] [1497643945.891433313]: hello world 9
```

Now let's run the listener:

```
$ publisher-subscriber.listener
```

And you'll see the this project works exactly the same as before:

```
[ INFO] [1497643969.443662208]: I heard: [hello world 41]
[ INFO] [1497643969.543668530]: I heard: [hello world 42]
[ INFO] [1497643969.643621679]: I heard: [hello world 43]
[ INFO] [1497643969.743650720]: I heard: [hello world 44]
[ INFO] [1497643969.843650108]: I heard: [hello world 45]
```

When you're done, as usual, press Ctrl-C to terminate roscore, as well as the talker and listener.

4. Sharing the snap with the world

Emailing snaps to people doesn't scale, but more importantly, it gives the snap users no upgrade path. If you make your snap available in the Ubuntu Store, whenever you release a new version your users will automatically update. In order to do this, you'll need to create a (free) store account at dashboard.snapcraft.io (<https://dashboard.snapcraft.io/>). Then, using Snapcraft, login with that account:

```
$ snapcraft login
```

Remember our previous discussion about how snap names need to be unique. Before you can release your snap in the store, you need to register the snap name to your store account:

```
$ snapcraft register <my snap name>
```

Finally, push the snap up to the store and release it into the stable channel:

```
$ snapcraft push path/to/my.snap --release=stable
```

Once the upload and automated reviews finish successfully, anyone in the world can install your snap as simply as:

```
$ sudo snap install <my snap name>
```

And they can rest secure knowing that when you release an update, they'll get it, no further work required.

Except where

otherwise noted,

the ROS wiki is

licensed under the

Creative Commons Attribution 3.0 (<http://creativecommons.org/licenses/by/3.0/>)

Wiki: ROS/Tutorials/Packaging your ROS project as a snap (dernière édition le 2018-10-18 16:54:04 par kyrofa (/kyrofa))

Brought to you by:  Open Source Robotics Foundation

(<http://www.osrfoundation.org>)