

GE5
2023

Tutoriel ROS2

Partie 2

Auteurs :

Arthur BOUILLÉ (arthur.bouille@insa-strasbourg.fr)
Alexandre THOUVENIN (alexandre.thouvenin@insa-strasbourg.fr)



Table des matières

| | | |
|----------|---|-----------|
| 1 | Introduction | 3 |
| 2 | Créer un workspace ROS2 | 4 |
| 2.1 | Mise de la source dans fichier <code>~/.bashrc</code> | 4 |
| 2.2 | Créer le répertoire qui servira de workspace | 5 |
| 2.3 | Construire votre workspace avec colcon | 6 |
| 3 | Comprendre les packages (Python et CMake) | 6 |
| 4 | Créer un package Python | 7 |
| 4.1 | Créer le package | 7 |
| 4.2 | Construire le package | 7 |
| 4.3 | Sourcer l'overlay de votre workspace | 8 |
| 4.4 | Tester votre premier package Python | 8 |
| 4.5 | Modifier votre premier package Python | 9 |
| 5 | Créer un package avec CMake | 10 |
| 5.1 | Créer le package | 10 |
| 5.2 | Construire le package | 10 |
| 5.3 | Sourcer l'overlay de votre workspace | 10 |
| 5.4 | Tester votre premier package CMake | 11 |
| 5.5 | Modifier votre premier package CMake | 11 |
| 6 | Créer d'un package publisher/subscriber en Python | 13 |
| 6.1 | Créer le package | 13 |
| 6.2 | Créer le noeud publisher | 14 |
| 6.3 | Explication du code du publisher | 15 |
| 6.4 | Créer le noeud subscriber | 16 |
| 6.5 | Explication du code du subscriber | 16 |
| 6.6 | Ajouter les dépendances pour le publisher | 17 |
| 6.7 | Ajouter les dépendances pour le subscriber | 19 |
| 6.8 | Construction du package | 19 |
| 6.9 | Sourcer l'overlay de votre workspace | 20 |
| 6.10 | Tester votre package Python | 20 |
| 6.11 | Visualiser les deux noeuds et le topic de votre package | 21 |
| 7 | Créer un package pour faire bouger une tortue de TurtleSim | 22 |
| 7.1 | Créer le package en Python | 22 |
| 7.2 | Créer l'exécutable python dans le package | 22 |
| 7.3 | Explication du code du publisher | 23 |
| 7.4 | Modifier les points d'entrée de votre package | 24 |
| 7.5 | Construire le package et sourcer l'overlay | 24 |
| 7.6 | Tester votre package Python | 24 |
| 8 | Bibliographie | 26 |

1 Introduction

Ce second tutoriel a pour but de comprendre plus en profondeur ce que sont les Publishers et les Subscribers. Notamment, de mettre en place une communication entre un Publisher et un Subscriber en utilisant le langage de programmation Python et CMake (C++). Nous allons donc d'abord nous intéresser à ce que sont Python et CMake.

- **Python** : Python est un langage de programmation simple, utilisé pour le développement back-end d'applications web ou mobile, mais aussi pour développer des applications pour PC. En plus de cela, Python possède un grand nombre de bibliothèque qui rendent son utilisation applicable dans la plupart des domaines, ce qui comble le fait que le codage d'applications lourdes devient très vite difficile.
- **CMake** : CMake est un système de construction logicielle multiplateforme. Il facilite grandement la création de projet, en vérifiant les prérequis nécessaires à leur création, et en déterminant les dépendances entre les différents composants d'un même projet.
On va alors pouvoir créer plusieurs packages qui servent à informer CMake des différentes dépendances, et ensuite utiliser des programmes écrits avec un langage de programmation supporté par CMake (C++ par exemple) dans chacun de ces packages. Cela, afin de créer un projet fonctionnel dont chaque partie est indépendamment géré par le système de construction. Il est alors plus facile de gérer et corriger des erreurs si elles apparaissent.

2 Créer un workspace ROS2

Un workspace est simplement un dossier qui contient des packages pour ROS 2. Avant d'utiliser ROS 2, vous devez sourcer les fichiers setup de ROS2 dans votre terminal. Cela se fait automatiquement car le fichier `.bashrc` contient les informations nécessaires. Cela rend les différentes fonctions de `ros2` disponibles pour une utilisation dans ce terminal. Cette source de ROS2 agit comme un **"underlay"** qui permet d'utiliser les fonctionnalités de ROS2.

Vous pouvez aussi sourcer un **"overlay"** - un espace de travail supplémentaire où vous pouvez ajouter de nouveaux packages, que vous avez créés ou non, sans perturber votre espace de travail ROS 2 actuel. Votre sous-couche (l'overlay) doit contenir les dépendances de tous les packages de votre "overlay". Les packages de votre "overlay" auront la priorité sur ceux de la sous-couche (c'est à dire que si vous avez un package qui modifie TurtleSim dans votre workspace et que vous sourcez l'overlay ; si vous lancez TurtleSim vous aurez alors sa version modifiée). Vous pouvez également avoir plusieurs couches de sous-couches et d'"overlays", et chaque "overlay" suivant utilise les packages de ses sous-couches parentes.

2.1 Mise de la source dans fichier `~/.bashrc`

Cette partie consiste à ce que la source de ROS2, "l'underlay" soit ajouté automatiquement pour chaque nouveau terminal que vous ouvrez. Pour ce faire suivez les instructions suivantes :

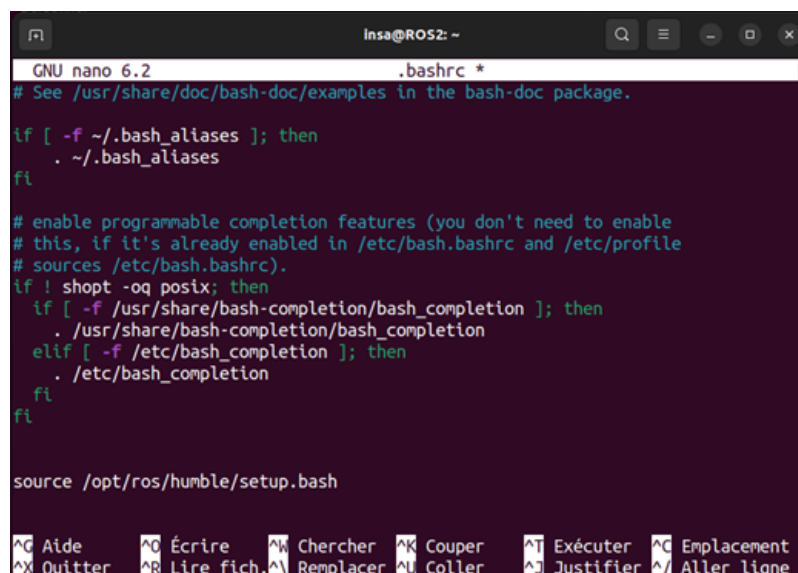
1. Ouvrez un terminal avec `"ctrl+alt+T"`, puis modifier le fichier `".bashrc"` avec la commande suivante puis tapez votre mot de passe :

```
insa@ROS2:~$ sudo nano .bashrc
[sudo] Mot de passe de insa :
```

La ligne de commande à été expliquée dans le **Tutoriel ROS2 : Partie 1**.

2. Indiquer le chemin du fichier setup de ROS2. Voici la ligne de code que vous avez à mettre à la fin de ce fichier : **`source /opt/ros/humble/setup.bash`**

Ajoutez le chemin du fichier setup de ROS2 à la fin de ce fichier et vous devriez obtenir :



```

GNU nano 6.2                                .bashrc *
# See /usr/share/doc/bash-doc/examples in the bash-doc package.

if [ -f ~/.bash_aliases ]; then
  . ~/.bash_aliases
fi

# enable programmable completion features (you don't need to enable
# this, if it's already enabled in /etc/bash.bashrc and /etc/profile
# sources /etc/bash.bashrc).
if ! shopt -oq posix; then
  if [ -f /usr/share/bash-completion/bash_completion ]; then
    . /usr/share/bash-completion/bash_completion
  elif [ -f /etc/bash_completion ]; then
    . /etc/bash_completion
  fi
fi

source /opt/ros/humble/setup.bash
  
```

Sauvegardez et quittez le fichier avec « `Ctrl + O` », « `Entrer` » et ensuite « `Ctrl + X` ».

La source du fichier `setup.bash` de ROS2 correspond à l'underlay de notre espace de travail. Par la suite nous ajouterons un overlay pour pouvoir utiliser nos propres packages.

2.2 Créer le répertoire qui servira de workspace

La meilleure pratique consiste à créer un nouveau répertoire pour chaque nouveau workspace. Le nom n'a pas d'importance, mais il est utile qu'il indique son objectif. Choisissons le nom de répertoire `ros2_ws`, pour "espace de développement".

1. Créez votre premier workspace en tapant la commande suivante :

```
insa@ROS2:~$ mkdir -p ~/ros2_ws/src
```

Explication de la commande :

`mkdir` : permet de créer un répertoire

`-p` : paramètre de la fonction `mkdir` permettant de créer directement un sous répertoires à notre répertoire principal

`~/ros2_ws/src` : définit un chemin de répertoire et sous répertoire.

Donc la commande `$ mkdir -p ~/ros2_ws/src` permet de créer le répertoire de notre workspace ainsi qu'un sous répertoire `src`

Une autre bonne pratique consiste à placer tous les packages de votre espace de travail dans le répertoire `src`.

2. Déplacez vous dans le répertoire `src` de votre workspace. Pour ce faire tapez la commande :

```
insa@ROS2:~$ cd ~/ros2_ws/src/
```

3. Vous pouvez utiliser la commande `ls` pour voir les fichiers dans le sous-répertoire `src`. Pour le moment, ce sous-répertoire est vide et cela est bien normal. C'est dans ce répertoire que vous allez créer prochainement des nouveaux packages.

4. Revenez dans la racine de votre workspace pour la suite du tutoriel (ie. revenez dans le répertoire `~/ros2_ws`).

Vous avez le choix entre deux méthodes pour faire ceci avec la commande `cd` (`cd => change directory`) :

- Première méthode (permet de choisir le chemin précis où vous souhaitez vous rendre) :

```
insa@ROS2:~/ros2_ws/src$ cd ~/ros2_ws/  
insa@ROS2:~/ros2_ws$
```

- Deuxième méthode (permet de revenir dans le répertoire parent de celui où vous êtes) :

```
insa@ROS2:~/ros2_ws/src$ cd ..  
insa@ROS2:~/ros2_ws$
```

5. Tapez la commande `ls` pour voir ce que contient votre workspace. Vous pourrez voir qu'il ne contient que votre sous répertoire `src` qui lui est vide pour le moment.

2.3 Construire votre workspace avec colcon

Ceci est un mini tutoriel sur la façon de construire un workspace ROS 2 avec colcon. Un workspace ROS2 est un répertoire avec une structure particulière. En général, il y a un sous-répertoire src (vous venez de le créer). A l'intérieur de ce sous-répertoire se trouve le code source des différents packages. Typiquement, le répertoire src commence par être vide (c'est votre cas!).

Depuis la racine de votre workspace (faites la commande `$ cd ~/ros2_ws`) pour aller dans la racine), vous pouvez maintenant construire vos paquets à l'aide de la commande colcon :

```
insa@ROS2:~/ros2_ws$ colcon build
```

Explication de colcon : colcon réalise la construction du workspace à partir des sources. Par défaut, il créera les répertoires suivants en tant que répertoire voisins de src :

- Le répertoire **build** sera l'endroit où les fichiers intermédiaires seront stockés. Pour chaque package, un sous-dossier sera créé dans lequel, par exemple, CMake sera invoqué.
- Le répertoire **install** est l'endroit où chaque package sera installé. Par défaut, chaque package est installé dans un sous-répertoire distinct.
- Le répertoire **log** contient diverses informations de chacune des invocation de colcon.

Vous pouvez ensuite faire un ls (en étant dans le répertoire du workspace ~/ros2_ws) pour voir que les répertoires build, install et log ont bien été créés dans la racine du workspace (à côté de src) :

```
insa@ROS2:~/ros2_ws$ ls
build install log src
```

3 Comprendre les packages (Python et CMake)

Un package est une unité d'organisation pour votre code ROS 2. Si vous voulez pouvoir installer votre code ou le partager avec d'autres, vous aurez besoin de l'organiser dans un package. Avec les packages, vous pouvez publier votre travail ROS 2 et permettre aux autres de le construire et de l'utiliser facilement.

ROS 2 utilise "**ament**" comme *système de construction* de packages et "**colcon**" comme *outil de construction* de ces packages. Vous pouvez créer un package en utilisant CMake ou Python, qui sont officiellement supportés, bien que d'autres types de construction existent. Les paquets ROS 2 Python et CMake ont chacun leur propre contenu minimum requis :

- Package Python
 - **package.xml** : fichier contenant des méta-informations sur le package
 - **resource/<nom_du_paquet>** : fichier marqueur pour le package
 - **setup.cfg** est nécessaire lorsqu'un package a des exécutables, afin que ros2 run puisse les trouver
 - **setup.py** contient les instructions pour l'installation du package
 - **<nom_du_paquet>** : est un répertoire avec le même nom que votre package, utilisé par les outils de ROS 2 pour trouver votre package, qui contient le fichier python `__init__.py`
- Package CMake
 - **CMakeLists.txt** : fichier qui décrit comment construire le code dans le package
 - **include/<nom_du_paquet>** : le répertoire contenant les en-têtes publics du package
 - **package.xml** : fichier contenant des méta-informations sur le package
 - **src** : répertoire contenant le code source du package

4 Créer un package Python

Dans cette partie nous allons créer un package python qui permet d'écrire du texte dans un terminal.

4.1 Créer le package

1. Déplacez vous dans le répertoire src de votre workspace. Pour ce faire tapez la commande :

```
insa@ROS2:~$ cd ~/ros2_ws/src/
```

2. Observez la structure de la commande à taper pour créer un package :

```
$ ros2 pkg create --build-type ament_python <package_name>
```

Nous appellerons notre package "my_package" et nous utiliserons l'argument optionnel - node-name qui crée un simple exécutable (nommé "my_node") de type Hello World dans le package.

3. Entrez la commande suivante dans votre terminal :

```
insa@ROS2:~/ros2_ws/src$ ros2 pkg create --build-type ament_python --node-name my_node my_package
```

Après avoir exécuté la commande, votre terminal affichera le message suivant :

```
insa@ROS2:~/ros2_ws/src$ ros2 pkg create --build-type ament_python --node-name my_node my_package
going to create a new package
package name: my_package
destination directory: /home/insa/ros2_ws/src
package format: 3
version: 0.0.0
description: TODO: Package description
maintainer: ['insa <insa@todo.todo>']
licenses: ['TODO: License declaration']
build type: ament_python
dependencies: []
node_name: my_node
creating folder ./my_package
creating ./my_package/package.xml
creating source folder
creating folder ./my_package/my_package
creating ./my_package/setup.py
creating ./my_package/setup.cfg
creating folder ./my_package/resource
creating ./my_package/resource/my_package
creating ./my_package/my_package/__init__.py
creating folder ./my_package/test
creating ./my_package/test/test_copyright.py
creating ./my_package/test/test_flake8.py
creating ./my_package/test/test_pep257.py
creating ./my_package/my_package/my_node.py
```

Vous aurez maintenant un nouveau dossier dans le répertoire src de votre espace de travail, appelé my_package.

4. Vous pouvez utiliser la commande *ls* pour voir les fichiers dans le sous-répertoire src. Vous constaterez que votre package a bien été créé :

```
insa@ROS2:~/ros2_ws/src$ ls
my_package
```

4.2 Construire le package

Le fait de placer les packages dans un workspace est particulièrement utile car vous pouvez ensuite construire plusieurs packages en même temps en lançant colcon build à la racine du workspace :

1. Retournez à la racine de votre espace de travail (Voir [2.2.4](#))

2. Construisez votre package en reconstruisant toute votre workspace en tapant la commande :

```
insa@ROS2:~/ros2_ws$ colcon build
```

Pour construire uniquement ce package et pas toute votre workspace, tapez la commande :

```
insa@ROS2:~/ros2_ws$ colcon build --packages-select my_package
```

4.3 Sourcer l'overlay de votre workspace

1. Votre package est donc construit et il peut donc être utilisé avec ROS2. Pour ce faire nous devons mettre la source de l'overlay :

```
insa@ROS2:~/ros2_ws$ source install/setup.bash
```

2. *N'oubliez pas cette étape sinon la commande suivante ne fonctionnera pas car "l'overlay" n'est pas sourcé.*

4.4 Tester votre premier package Python

1. Observez la structure de la commande permettant de lancer le package :

```
$ ros2 run <package_name> <node_name>
```

Dans notre cas, le package se nomme "my_package". <node_name> correspond à l'exécutable du package : "my_node". Cet exécutable a été créé automatiquement lors de la création du package (à cause du paramètre "- node-name my_node").

2. Testez le package en tapant la commande :

```
insa@ROS2:~/ros2_ws$ ros2 run my_package my_node
```

Vous obtiendrez dans le terminal :

```
insa@ROS2:~/ros2_ws$ ros2 run my_package my_node
Hi from my_package.
```

Vous pouvez voir que l'exécutable "my_node" du package permet de faire afficher du texte dans le terminal.

3. Suivez les instructions suivantes, pour modifier la phrase retournée dans le terminal. Rendez-vous dans répertoire ~/ros2_ws/src/my_package/my_package/ :

```
insa@ROS2:~/ros2_ws$ cd ~/ros2_ws/src/my_package/my_package/
insa@ROS2:~/ros2_ws/src/my_package/my_package$
```

Faites la commande `ls` :

```
insa@ROS2:~/ros2_ws/src/my_package/my_package$ ls
__init__.py  my_node.py
```

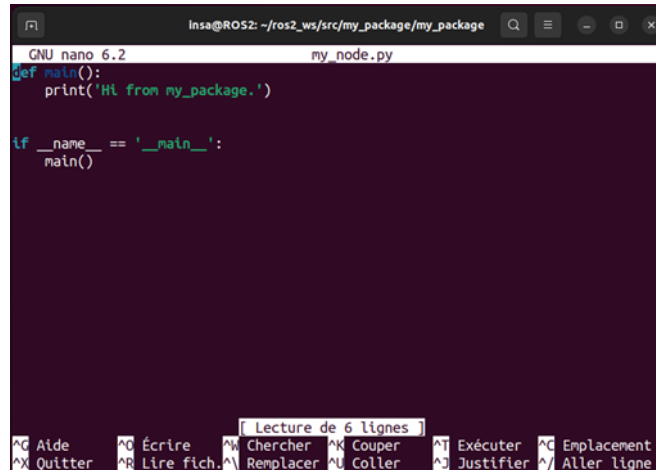
Le fichier `__init__.py` est un fichier créé automatiquement lors du build du package.
Le fichier `my_node.py` est l'exécutable qui a été créé automatiquement lors de la création du package. Cet exécutable est un code python servant, ici, à afficher du texte dans le terminal.

4.5 Modifier votre premier package Python

1. Ouvrez le fichier `my_node.py` avec le même type de commande que pour modifier le fichier `.bashrc`. Pour rappel, utiliser la fonction `$ sudo nano <nom_fichier>` :

```
insa@ROS2:~/ros2_ws/src/my_package/my_package$ sudo nano my_node.py
```

Le code python `my_node.py` va s'ouvrir avec l'éditeur nano :



```

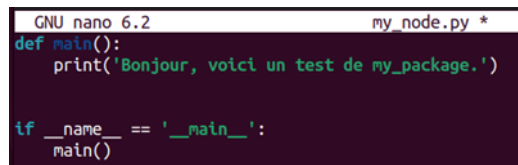
GNU nano 6.2 my_node.py
def main():
    print('Hi from my_package.')

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

At the bottom of the editor, a status bar shows: "Lecture de 6 lignes" and a list of keyboard shortcuts: Aide, Écrire, Chercher, Couper, Exécuter, Emplacement, Quitter, Lire fich., Remplacer, Coller, Justifier, Aller ligne.

En observant le code vous pouvez voir la présence d'une condition qui est toujours valide qui appelle la fonction `main()`. Cette fonction `main()` contient la fonction `print()` qui affiche le texte sur terminal.

2. Modifiez la phrase présente dans la fonction `print()`. Mettez par exemple :



```

GNU nano 6.2 my_node.py *
def main():
    print('Bonjour, voici un test de my_package.')

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

Enregistrez le fichier ! A cette étape, vous avez besoin de reconstruire juste votre package et pas tous votre workspace.

3. **Retournez dans la racine** et tapez la commande :

```
insa@ROS2:~/ros2_ws$ colcon build --packages-select my_package
```

4. Tapez la commande suivante, et vérifiez que le texte affiché a bien été modifié lors de la construction de votre package :

```
insa@ROS2:~/ros2_ws$ ros2 run my_package my_node
Bonjour, voici un test de my package.
```

Voilà, vous avez créé un package simple en python , permettant de créer un nœud « `my_node` » qui affiche une ligne de texte dans un terminal.

5 Créer un package avec CMake

Dans cette partie nous allons créer un package avec CMake qui permet d'écrire du texte dans un terminal.

5.1 Créer le package

Nous allons créer le même package qu'en python mais cette fois-ci nous allons utiliser CMake. La démarche est quasiment similaire.

1. Rendez vous dans le répertoire src de votre workspace. Pour ce faire tapez la commande :

```
insa@ROS2:~$ cd ~/ros2_ws/src/
```

2. Observez la structure de la commande à taper pour créer un package en Cmake. Elle est similaire à celle pour le python :

```
$ ros2 pkg create --build-type ament_cmake <package_name>
```

Nous appellerons notre package "my_package2" et nous utiliserons l'argument optionnel - node-name qui crée un simple exécutable (nommé "my_node") de type Hello World dans le package.

3. Entrez la commande suivante dans votre terminal, pour créer un package nommé my_package2 en CMake :

```
insa@ROS2:~/ros2_ws/src$ ros2 pkg create --build-type ament_cmake --node-name my_node my_package2
```

Vous aurez maintenant un nouveau dossier dans le répertoire src de votre espace de travail appelé my_package2.

4. Vous pouvez utiliser la commande `ls` pour voir les fichiers dans le sous-répertoire src de votre workspace. Vous constaterez que votre package a bien été créé.

5.2 Construire le package

Le fait de placer les packages dans un workspace est particulièrement utile car vous pourrez ensuite construire plusieurs packages en même temps en lançant `colcon build` à la racine du workspace.

1. Retournez à la racine de votre espace de travail (Voir 2.2.4)
2. Construisez votre package en reconstruisant uniquement, **my_package2**, votre nouveau package :

```
insa@ROS2:~/ros2_ws$ colcon build --packages-select my_package2
Starting >>> my_package2
Finished <<< my_package2 [2.04s]

Summary: 1 package finished [3.49s]
```

Si vous souhaitez faire la reconstruction de votre workspace entier, suivez les commandes de la section 4.2.2

5.3 Sourcer l'overlay de votre workspace

1. Votre package est donc construit et peut donc être utilisé avec ROS2. Pour ce faire nous devons mettre la source de l'overlay :

```
insa@ROS2:~/ros2_ws$ source install/setup.bash
```

2. *N'oubliez pas cette étape sinon la commande suivante ne fonctionnera pas car vous n'aurez pas sourcé "l'overlay".*

5.4 Tester votre premier package CMake

1. Observez la structure de la commande permettant de lancer votre package :

```
$ ros2 run <package_name> <node_name>
```

Dans notre cas, notre package se nomme "my_package2". <node_name> correspond à l'exécutable de notre package : "my_node". Cet exécutable a été créé automatiquement lors de la création du package (à cause du paramètre "- -node-name my_node").

2. Testez votre package en tapant la commande :

```
insa@ROS2:~/ros2_ws$ ros2 run my_package2 my_node
```

Vous obtiendrez dans le terminal :

```
insa@ROS2:~/ros2_ws$ ros2 run my_package2 my_node
hello world my_package2 package
```

Vous pouvez voir que l'exécutable "my_node" de votre package permet de faire afficher du texte dans le terminal comme le package en python.

3. Suivez les instructions suivantes, pour modifier la phrase retournée dans le terminal. Rendez-vous dans répertoire ~/ros2_ws/src/my_package2/src/ (Rappel : \$ cd ~/ros2_ws/src/my_package2/src/) et faites ensuite la commande ls :

```
insa@ROS2:~/ros2_ws/src/my_package2/src$ ls
my_node.cpp
```

Le fichier my_node.cpp est l'exécutable qui a été créé automatiquement lors de la création du package. Cet exécutable est un code en c++ servant, ici, à afficher du texte dans le terminal.

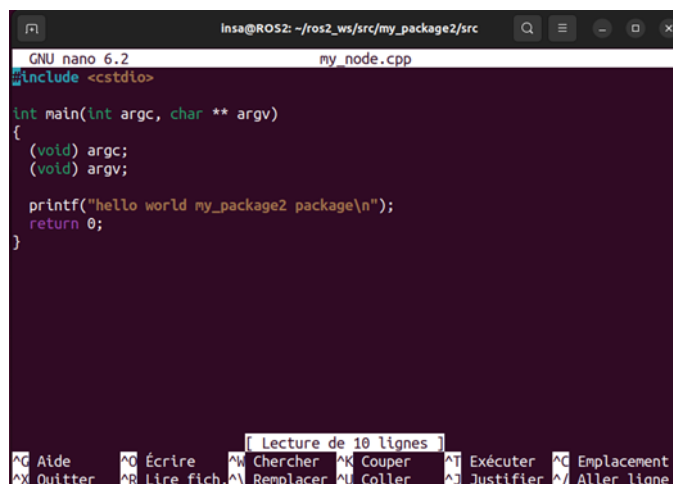
Contrairement à un package python, vous pouvez voir que la structure du package en CMake est différente de celle d'un package en python (Comme on vous l'a montré à la [section 3](#))

5.5 Modifier votre premier package CMake

1. Ouvrez le fichier my_node.cpp avec le même type de commande que pour modifier le fichier .bashrc. Pour rappel, utiliser la fonction \$ sudo nano <nom_fichier> :

```
insa@ROS2:~/ros2_ws/src/my_package2/src$ sudo nano my_node.cpp
```

Le code python my_node.cpp va s'ouvrir avec l'éditeur nano :



```
GNU nano 6.2 my_node.cpp
#include <stdio>

int main(int argc, char ** argv)
{
    (void) argc;
    (void) argv;

    printf("hello world my_package2 package\n");
    return 0;
}
```

Vous constaterez directement que le code est totalement différent d'un code python. Ici nous avons uniquement une fonction main qui contient la fonction printf() avec la phrase que vous avez pu voir dans votre terminal précédemment.

2. Modifiez le texte présent dans la fonction printf(). Mettez par exemple :

```
GNU nano 6.2 my_node.cpp *
#include <stdio>

int main(int argc, char ** argv)
{
    (void) argc;
    (void) argv;

    printf("Bonjour, voici un test de my_package2 en cpp !\n");
    return 0;
}
```

Sauvegardez avec "ctrl+O", puis "Entrer" et "ctrl+X".

A cette étape, vous avez besoin de reconstruire uniquement votre package et pas tous votre workspace.

3. **Retournez dans la racine** et tapez la commande :

```
insa@ROS2:~/ros2_ws$ colcon build --packages-select my_package2
Starting >>> my_package2
Finished <<< my_package2 [2.04s]

Summary: 1 package finished [3.49s]
```

4. Tapez la commande suivante pour lancer votre package, et vérifiez que le texte affiché a été bien modifié lors de la construction de votre package :

```
insa@ROS2:~/ros2_ws$ ros2 run my_package2 my_node
Bonjour, voici un test de my_package2 en cpp !
```

Voilà, vous avez créé un package simple en CMake , permettant de créer un nœud « my_node » qui affiche une ligne de texte dans un terminal.

6 Créer d'un package publisher/subscriber en Python

Dans ce tutoriel, vous allez créer des nœuds qui se transmettent des informations sous la forme de messages de chaînes de caractères par l'intermédiaire d'un topic. L'exemple utilisé ici est un simple système "talker" et "listener" ; un nœud publie des données sur le topic et l'autre s'abonne au topic afin de recevoir ces données.

6.1 Créer le package

1. Rendez-vous dans le répertoire src de votre workspace "ros2_ws/src". (Voir section 5.1.1)
2. Créez un package python que vous nommerez py_pubsub avec la commande suivante :

```
insa@ROS2:~/ros2_ws/src$ ros2 pkg create --build-type ament_python py_pubsub
```

Après avoir lancé la commande, votre terminal vous renverra le message suivant :

```
insa@ROS2:~/ros2_ws/src$ ros2 pkg create --build-type ament_python py_pubsub
going to create a new package
package name: py_pubsub
destination directory: /home/insa/ros2_ws/src
package format: 3
version: 0.0.0
description: TODO: Package description
maintainer: ['insa <insa@todo.todo>']
licenses: ['TODO: License declaration']
build type: ament_python
dependencies: []
creating folder ./py_pubsub
creating ./py_pubsub/package.xml
creating source folder
creating folder ./py_pubsub/py_pubsub
creating ./py_pubsub/setup.py
creating ./py_pubsub/setup.cfg
creating folder ./py_pubsub/resource
creating ./py_pubsub/resource/py_pubsub
creating ./py_pubsub/py_pubsub/__init__.py
creating folder ./py_pubsub/test
creating ./py_pubsub/test/test_copyright.py
creating ./py_pubsub/test/test_flake8.py
creating ./py_pubsub/test/test_pep257.py

[WARNING]: Unknown license 'TODO: License declaration'. This has been set in the package.xml, but no L
ICENSE file has been created.
It is recommended to use one of the ament license identifiers:
Apache-2.0
BSL-1.0
BSD-2.0
BSD-2-Clause
BSD-3-Clause
GPL-3.0-only
LGPL-3.0-only
MIT
MIT-0
```

Vous avez peut-être remarqué dans le message après la création de votre package que les champs description et licence contiennent des notes TODO.

C'est parce que la description du package et la déclaration de licence ne sont pas automatiquement définies, mais sont nécessaires si vous souhaitez publier votre package.

Dans ce tutoriel, vous allez aussi apprendre à modifier ces informations.

3. Utilisez la commande `ls` pour voir les fichiers dans le sous-répertoire src. Vous constaterez que votre package « py_pubsub » a bien été créé.
Nous devons maintenant écrire les nœuds correspondants au publisher et au subscriber dans le répertoire ~/ros2_ws/src/py_pubsub/py_pubsub/

6.2 Créer le noeud publisher

1. Rendez-vous dans le répertoire `~/ros2_ws/src/py_pubsub/py_pubsub/` :

```
lnsa@ROS2: ~/ros2_ws/src$ cd py_pubsub/py_pubsub/
```

2. Créez un fichier « `publisher_member_function.py` » avec la commande

```
lnsa@ROS2: ~/ros2_ws/src/py_pubsub/py_pubsub$ sudo nano publisher_member_function.py
```

3. Ecrivez les lignes suivantes dans le fichier que l'éditeur a ouvert :



```
GNU nano 6.2 publisher_member_function.py *
import rclpy
from rclpy.node import Node

from std_msgs.msg import String

class MinimalPublisher(Node):

    def __init__(self):
        super().__init__('minimal_publisher')
        self.publisher_ = self.create_publisher(String, 'topic', 10)
        timer_period = 0.5 # seconds
        self.timer = self.create_timer(timer_period, self.timer_callback)
        self.i = 0

    def timer_callback(self):
        msg = String()
        msg.data = 'Hello World: %d' % self.i
        self.publisher_.publish(msg)
        self.get_logger().info('Publishing: "%s"' % msg.data)
        self.i += 1

def main(args=None):
    rclpy.init(args=args)

    minimal_publisher = MinimalPublisher()

    rclpy.spin(minimal_publisher)

    # Destroy the node explicitly
    # (optional - otherwise it will be done automatically
    # when the garbage collector destroys the node object)
    minimal_publisher.destroy_node()
    rclpy.shutdown()

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

- Faites attention à l'indentation que vous utilisez dans votre code : Utilisez soit des espaces soit des tabulations mais pas les deux dans le même code.
- Faites attention à `__init__` qui contient 2 underscore avant et après le init. Pareil pour `__name__` et `__main__`.

Sauvegardez avec "ctrl+O", puis "Entrer" et "ctrl+X".

6.3 Explication du code du publisher

1. Les premières lignes de code permettent d'importer rclpy pour que la classe *Node* puisse être utilisée :

```
import rclpy
from rclpy.node import Node
```

La déclaration suivante importe le type de message intégré que le nœud utilisera pour structurer les données qu'il transmettra au topic. Soit, des messages de type *String* :

```
from std_msgs.msg import String
```

Ces lignes représentent les dépendances du nœud. Rappelez-vous que les dépendances doivent être ajoutées au fichier package.xml, ce que vous ferez dans la section suivante.

2. Ensuite, la classe *MinimalPublisher* est créée, qui hérite de (ou est une sous-classe de) *Node* :

```
class MinimalPublisher(Node):
```

3. Voici, ci-dessous, le constructeur de la classe. La fonction *super().__init__* appelle le constructeur de la classe *Node* et lui donne le nom de notre nœud, dans ce cas "minimal_publisher".

create_publisher déclare que le nœud publie des messages de type *String* (importé du module *std_msgs.msg*), sur un topic nommé "topic", et que la "taille de la file d'attente" est de 10. La taille de la file d'attente est un paramètre de QoS (qualité de service) requis qui limite le nombre de messages mis en file d'attente si un subscriber ne les reçoit pas assez rapidement.

Ensuite, un timer est créé avec un callback à exécuter toutes les 0,5 secondes. *self.i* est un compteur utilisé dans le callback :

```
def __init__(self):
    super().__init__('minimal_publisher')
    self.publisher_ = self.create_publisher(String, 'topic', 10)
    timer_period = 0.5 # seconds
    self.timer = self.create_timer(timer_period, self.timer_callback)
    self.i = 0
```

4. *timer_callback* crée un message avec la valeur du compteur en annexe, et le publie sur le terminal avec la fonction *get_logger().info()*.

```
def timer_callback(self):
    msg = String()
    msg.data = 'Hello World: %d' % self.i
    self.publisher_.publish(msg)
    self.get_logger().info('Publishing: "%s"' % msg.data)
    self.i += 1
```

5. Enfin, la fonction principale, *main()*, est définie :

```
def main(args=None):
    rclpy.init(args=args)

    minimal_publisher = MinimalPublisher()


    rclpy.spin(minimal_publisher)

    # Destroy the node explicitly
    # (optional - otherwise it will be done automatically
    # when the garbage collector destroys the node object)
    minimal_publisher.destroy_node()
    rclpy.shutdown()
```

La bibliothèque rclpy est d'abord initialisée, puis le nœud est créé avec la class *MinimalPublisher()*, et enfin le code fait "spin" le nœud pour que ses callbacks soient appelés toutes les 0,5s.

6.4 Créer le noeud subscriber

1. Rendez-vous dans le répertoire `~/ros2_ws/src/py_pubsub/py_pubsub/` et créez un fichier `subscriber_member_function.py` avec la commande **sudo nano** (Voir la section 6.2.1/6.2.2).
2. Ecrivez les lignes suivantes dans le fichier que l'éditeur a ouvert :



```

GNU nano 6.2 subscriber_member_function.py *
import rclpy
from rclpy.node import Node

from std_msgs.msg import String

class MinimalSubscriber(Node):

    def __init__(self):
        super().__init__('minimal_subscriber')
        self.subscription = self.create_subscription(
            String,
            'topic',
            self.listener_callback,
            10)
        self.subscription # prevent unused variable warning

    def listener_callback(self, msg):
        self.get_logger().info('I heard: "%s"' % msg.data)

def main(args=None):
    rclpy.init(args=args)

    minimal_subscriber = MinimalSubscriber()

    rclpy.spin(minimal_subscriber)

    # Destroy the node explicitly
    # (optional - otherwise it will be done automatically
    # when the garbage collector destroys the node object)
    minimal_subscriber.destroy_node()
    rclpy.shutdown()

if __name__ == '__main__':
    main()

```

- Faites attention à l'indentation que vous utilisez dans votre code : Utilisez soit des espaces soit des tabulations mais pas les deux dans le même code.
- Faites attention à `__init__` qui contient 2 underscore avant et après le init. Pareil pour `__name__` et `__main__`.

Sauvegardez avec "ctrl+O", puis "Entrer" et "ctrl+X".

6.5 Explication du code du subscriber

1. Le code du nœud subscriber est presque identique à celui du publisher. Le constructeur crée un subscriber avec les mêmes arguments que le publisher. Rappelons que le nom du topic et aussi le type de message utilisés par le publisher et le subscriber doivent correspondre pour qu'ils puissent communiquer.

```

self.subscription = self.create_subscription(
    String,
    'topic',
    self.listener_callback,
    10)

```

2. Le constructeur et le callback du subscriber n'incluent pas de définition de timer, car il n'en a pas besoin. Son callback est appelé dès qu'il reçoit un message.

La définition du callback imprime simplement un message d'information sur la console, avec les données qu'il a reçues. Rappelons que le publisher définit : `msg.data = 'Hello World : %d' % self.i` :

```
def listener_callback(self, msg):
    self.get_logger().info('I heard: "%s"' % msg.data)
```

La définition principale 'main()' est presque exactement la même, remplaçant la création et le "spin" du publisher par celui du subscriber.

Voilà vous venez de créer deux noeuds (un publisher et un subscriber) dans le même package (py_pubsub) !

6.6 Ajouter les dépendances pour le publisher

Dans cette section, nous allons ajouter des dépendances comme la description de notre package, l'email de contact, votre nom, et une licence.

1. Rendez-vous dans le répertoire `~/ros2_ws/src/py_pubsub/` (*rappel* : `$ cd ~/ros2_ws/src/py_pubsub/`)
2. Faites `ls` pour voir la liste des fichiers et répertoires disponible dans votre package `py_pubsub` :

```
insa@ROS2:~/ros2_ws/src/py_pubsub$ ls
package.xml  py_pubsub  resource  setup.cfg  setup.py  test
```

Vous pouvez voir que les différents fichiers d'un package python, explicités dans la [section 3](#), sont présents.

Dans notre cas, nous allons commencer par modifier le fichier "package.xml".

3. Ouvrez le fichier « package.xml » avec la commande **sudo nano** et modifier les lignes `<description>`, `<maintainer>` et `<license>` avec vos informations personnelles :

```
<description>Package Publisher and Subscriber</description>
<maintainer email="arthur.bouille@insa-strasbourg.fr">Arthur BOUILLE</maintainer>
<license>Apache License 2.0</license>
```

A la suite de ces lignes, ajoutez les dépendances correspondantes aux imports de notre nœud publisher :

```
<exec_depend>rclpy</exec_depend>
<exec_depend>std_msgs</exec_depend>
```

Votre fichier "package.xml" devrait ressembler à ceci :

```
GNU nano 6.2 package.xml *
<?xml version="1.0"?>
<?xml-model href="http://download.ros.org/schema/package_format3.xsd" schematypens="http://www.w3.org/2001/XMLSchema-instance" />
<package format="3">
  <name>py_pubsub</name>
  <version>0.0.0</version>
  <description>Package Publisher and Subscriber</description>
  <maintainer email="arthur.bouille@insa-strasbourg.fr">Arthur BOUILLE</maintainer>
  <license>Apache License 2.0</license>

  <exec_depend>rclpy</exec_depend>
  <exec_depend>std_msgs</exec_depend>

  <test_depend>ament_copyright</test_depend>
  <test_depend>ament_flake8</test_depend>
  <test_depend>ament_pep257</test_depend>
  <test_depend>python3-pytest</test_depend>

  <export>
    <build_type>ament_python</build_type>
  </export>
</package>
```

Sauvegardez le fichier !

4. Modifiez ensuite le fichier setup.py avec la commande `sudo nano`.
 5. Faites correspondre les champs `maintainer`, `maintainer_email`, `description` et `license` avec ceux du fichier `package.xml` (modifié précédemment).
 6. Ajoutez un point d'entrée pour le nœuds publisher entre les crochet de '`console_script`'. Pour ce faire vous devrez ajouter la ligne `'talker = py_pubsub.publisher_member_function :main'`, entre les crochet de '`console_script`'.
- Votre fichier setup.py devrait être identique à celui ci-dessous :

```

GNU nano 6.2                                setup.py *
from setuptools import find_packages, setup

package_name = 'py_pubsub'

setup(
    name=package_name,
    version='0.0.0',
    packages=find_packages(exclude=['test']),
    data_files=[
        ('share/ament_index/resource_index/packages',
         ['resource/' + package_name]),
        ('share/' + package_name, ['package.xml']),
    ],
    install_requires=['setuptools'],
    zip_safe=True,
    maintainer='Arthur BOUILLE',
    maintainer_email='arthur.bouille@insa-strasbourg.fr',
    description='Package Publisher and Subscriber',
    license='Apache License 2.0',
    tests_require=['pytest'],
    entry_points={
        'console_scripts': [
            'talker = py_pubsub.publisher_member_function:main',
        ],
    },
)

```

Sauvegarder le fichier !

7. Vérifiez que le fichier setup.cfg a bien été créé correctement.
 - Ouvrez le fichier setup.cfg
 - Vous devriez obtenir :

```

GNU nano 6.2                                setup.cfg
[develop]
script_dir=$base/lib/py_pubsub
[install]
install_scripts=$base/lib/py_pubsub

```

Il s'agit simplement d'indiquer à setuptools de placer vos exécutables dans lib, car la fonction `run` de `ros2` les cherchera à cet emplacement.

Toutes les dépendances du publisher ont été ajoutés. Passons aux dépendances du subscriber.

6.7 Ajouter les dépendances pour le subscriber

Les fichiers modifiés précédemment sont les mêmes pour nos deux nœuds (publisher et subscriber).

1. Comme les dépendances correspondantes aux imports de nos 2 nœuds sont identiques ('roscpp' et 'std_msgs'), nous avons uniquement besoin de modifier le fichier setup.py. en ajoutant un point d'entrée pour le subscriber.
2. Ajoutez la ligne 'listener = py_pubsub.subscriber_member_function :main', entre les crochets de 'console_script' et après la ligne d'entrée du publisher.
3. Vous devriez obtenir :

```
entry_points={
    'console_scripts': [
        'talker = py_pubsub.publisher_member_function:main',
        'listener = py_pubsub.subscriber_member_function:main',
    ],
}
```

Sauvegardez le fichier !

Toutes les dépendances ont été mises dans les différents fichiers. Passons à la construction du package py_pubsub contenant les 2 nœuds.

6.8 Construction du package

1. Revenez dans la racine de votre workspace (*Rappel* : `$ cd ~/ros2_ws/`) :
2. Avant de construire votre package, lancez une fonction qui permet de télécharger les bibliothèques des différentes dépendances que vous avez utilisées dans vos différents packages et qui ne sont pas encore téléchargées. La procédure se décompose en 3 étapes :

- Tapez la commande suivante :

```
insa@ROS2:~/ros2_ws$ sudo rosdep init
```

Il est possible que cela prenne plusieurs minutes ! À la fin, un message vous avertira qu'il est préférable de taper la commande suivante.

- Faire la mise à jour des dépendances :

```
insa@ROS2:~/ros2_ws$ rosdep update
```

Il est possible que cela prenne plusieurs minutes !

- Tapez une commande pour faire une vérification des dépendances :

```
insa@ROS2:~/ros2_ws$ rosdep install -i --from-path src --rosdistro humble -y
#All required rosdeps installed successfully
```

Utiliser ces 3 commandes est une bonne habitude à prendre car vous risquez d'avoir des packages "buggés" qui se construisent correctement mais qui seront inutilisables en cas de dépendance manquante !

3. Construisez votre package uniquement et pas tous votre workspace en tapant la commande :

```
insa@ROS2:~/ros2_ws$ colcon build --packages-select py_pubsub
```

4. Faites un `ls` pour vérifier que votre package 'py_pubsub' se trouve bien dans le répertoire ~/ros_ws/src/

6.9 Sourcer l'overlay de votre workspace

1. Votre package est donc construit et peut donc être utilisé avec ROS2. Avant de mettre à source de l'overlay dans un nouveau terminal, soyez à la racine de votre workspace (`$ cd ~/ros_ws`) comme précédemment puis tapez la commande :

```
insa@ROS2:~/ros2_ws$ source install/setup.bash
```

2. *N'oubliez pas cette étape sinon la commande suivante ne fonctionnera pas car "l'overlay" ne sera pas sourcé.*

6.10 Tester votre package Python

1. Dans notre cas, notre package se nomme "py_pubsub".
Il contient les exécutables "publisher_member_function.py" et "subscriber_member_function.py". Cependant, les noeuds se nomment "talker" et "listener". Ces noms viennent directement du fichier setup.py et des noms des points d'entrées de notre package py_pubsub :

```
entry_points={
  'console_scripts': [
    'talker = py_pubsub.publisher_member_function:main',
    'listener = py_pubsub.subscriber_member_function:main',
  ],
}
```

2. Testez le noeud publisher, nommé "talker", de votre package en tapant la commande suivante, dans le premier terminal :

```
insa@ROS2:~/ros2_ws$ ros2 run py_pubsub talker
```

Vous obtiendrez dans le terminal :

```
insa@ROS2:~/ros2_ws$ ros2 run py_pubsub talker
[INFO] [1697104476.968817129] [minimal_publisher]: Publishing: "Hello World: 0"
[INFO] [1697104477.466467277] [minimal_publisher]: Publishing: "Hello World: 1"
[INFO] [1697104477.969300519] [minimal_publisher]: Publishing: "Hello World: 2"
[INFO] [1697104478.469812089] [minimal_publisher]: Publishing: "Hello World: 3"
[INFO] [1697104478.962847748] [minimal_publisher]: Publishing: "Hello World: 4"
[INFO] [1697104479.460938906] [minimal_publisher]: Publishing: "Hello World: 5"
[INFO] [1697104479.961138939] [minimal_publisher]: Publishing: "Hello World: 6"
[INFO] [1697104480.461768362] [minimal_publisher]: Publishing: "Hello World: 7"
[INFO] [1697104480.965807305] [minimal_publisher]: Publishing: "Hello World: 8"
[INFO] [1697104481.461206734] [minimal_publisher]: Publishing: "Hello World: 9"
[INFO] [1697104481.964381574] [minimal_publisher]: Publishing: "Hello World: 10"
[INFO] [1697104482.460845811] [minimal_publisher]: Publishing: "Hello World: 11"
```

Vous pouvez voir que noeud "talker" de votre package permet de faire afficher du texte dans le terminal à un intervalle régulier de 0,5s.

3. Ouvrez un 2ème terminal et allez dans la racine de votre workspace (`$ cd ~/ros2_ws`). Tapez, ensuite, la commande pour **mettre la source de l'overlay** dans le terminal (Voir 6.9.1). Puis, tapez la commande pour lancer le noeuds du subscriber, nommé "listener" :

```
insa@ROS2:~/ros2_ws$ ros2 run py_pubsub listener
```

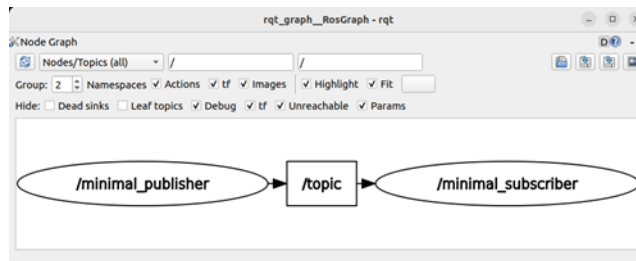
Vous obtiendrez dans le 3ème terminal :

```
insa@ROS2:~/ros2_ws$ ros2 run py_pubsub listener
[INFO] [1697104711.807150833] [minimal_subscriber]: I heard: "Hello World: 5"
[INFO] [1697104712.298714286] [minimal_subscriber]: I heard: "Hello World: 6"
[INFO] [1697104712.799437445] [minimal_subscriber]: I heard: "Hello World: 7"
[INFO] [1697104713.304721009] [minimal_subscriber]: I heard: "Hello World: 8"
[INFO] [1697104713.799061285] [minimal_subscriber]: I heard: "Hello World: 9"
[INFO] [1697104714.298930886] [minimal_subscriber]: I heard: "Hello World: 10"
[INFO] [1697104714.798761333] [minimal_subscriber]: I heard: "Hello World: 11"
[INFO] [1697104715.298912449] [minimal_subscriber]: I heard: "Hello World: 12"
```

Vous avez créé deux nœuds. Un pour publier et l'autre pour souscrire à ces données sur un topic. Avant de les exécuter, vous avez ajouté leurs dépendances et leurs points d'entrées aux fichiers de configuration du package.

6.11 Visualiser les deux noeuds et le topic de votre package

1. Visualisez vos deux nœuds communiquer via un topic grâce à `rqt_graph`.
Ouvrez un 3ème terminal et tapez la commande pour ouvrir `rqt_graph`. (Rappel de la commande : `$ rqt_graph`)
2. Vous obtiendrez le graphe suivant dans `rqt_graph` :



Le tutoriel est fini ! Vous savez maintenant créer plusieurs noeuds dans un seul package python. Pour le faire en CMake la méthode est similaire (langage en cpp et structure différentes), mais nous ne la traiterons pas ici.

7 Créer un package pour faire bouger une tortue de TurtleSim

Dans cette partie nous allons créer un package python qui permet de faire bouger une tortue de TurtleSim. Nous commencerons par la création du package.

7.1 Créer le package en Python

1. Déplacez vous dans le répertoire src de votre workspace. Pour ce faire tapez la commande :

```
insa@ROS2:~$ cd ~/ros2_ws/src/
```

Nous appellerons notre package : "my_turtle_controller".

2. Entrez, donc, la commande suivante dans votre terminal :

```
insa@ROS2:~/ros2_ws/src$ ros2 pkg create --build-type ament_python my_turtle_controller
```

Vous aurez maintenant un nouveau dossier dans le répertoire src de votre espace de travail appelé my_turtle_controller.

3. Vérifiez que ce package a bien été créé, en utilisant la commande `ls` dans le sous-répertoire src (Rappel : `$ cd ~/ros2_ws/src` et ensuite `$ ls`). Vous constaterez que votre package a bien été créé.

7.2 Créer l'exécutable python dans le package

Dans cette partie nous allons créer le fichier python qui sera un exécutable du package. Le code de ce fichier permettra de publier des messages de type `geometry_msgs/msg/Twist` sur le topic `/turtle1/cmd_vel` de TurtleSim. C'est à dire que nous allons publier des message contenant des vitesses linéaire et angulaire sur le topic `/turtle1/cmd_vel`, mais cette fois ci en utilisant python dans le but que la tortue bouge en forme de cercle.

1. Rendez-vous dans le répertoire "`~/ros2_ws/src/my_turtle_controller/my_turtle_controller`".
2. Créez un fichier que nous nommerons "`draw_circle.py`" :

```
insa@ROS2:~/ros2_ws/src/my_turtle_controller/my_turtle_controller$ sudo nano draw_circle.py
```

3. Écrivez le code suivant dans la fenêtre de l'éditeur de texte :

```

GNU nano 6.2 draw_circle.py
import rclpy
from rclpy.node import Node
from geometry_msgs.msg import Twist

class DrawCircleNode(Node):
    def __init__(self):
        super().__init__('draw_circle')
        self.cmd_vel_pub = self.create_publisher(Twist, "/turtle1/cmd_vel", 10)
        timer_period = 0.5 # seconds
        self.timer = self.create_timer(timer_period, self.send_velocity_command)
        self.get_logger().info("Draw circle node has been started")

    def send_velocity_command(self):
        msg = Twist()
        msg.linear.x = 2.0
        #msg.linear.y = 1.0
        msg.angular.z = 1.0
        self.cmd_vel_pub.publish(msg)

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

```

Comme vous le constatez peut-être, ce code ressemble fortement au code du publisher du précédent tutoriel. La seule différence c'est que les messages transmis sont ici du type `geometry_msgs/msg/Twist` (une structure sous la forme de deux vecteurs, un pour les vitesses linéaires et l'autre pour les vitesses angulaires).

- **Faites attention à l'indentation que vous utilisez dans votre code : Utilisez soit des espaces soit des tabulations mais pas les deux dans le même code.**
- **Faites attention à `__init__` qui contient 2 underscore avant et après le init. Pareil pour `__name__` et `__main__`.**

Sauvegarder avec "ctrl+O", puis "Entrer" et "ctrl+X".

7.3 Explication du code du publisher

1. Les premières lignes de code après les commentaires permettent d'importer `rclpy` pour que sa classe `Node` puisse être utilisée :

```
import rclpy
from rclpy.node import Node
```

La déclaration suivante importe le type de message `Twist` que le nœud utilise pour structurer les données qu'il transmet au topic `/turtle1/cmd_vel` :

```
from geometry_msgs.msg import Twist
```

2. Ensuite, la classe `DrawCircleNode` est créée et hérite de `Node` :

```
class DrawCircleNode(Node):
```

3. Voici, ci-dessous, le constructeur de la classe. La fonction `super().__init__` appelle le constructeur de la classe `Node` et lui donne le nom de votre nœud, dans ce cas `"draw_circle"`.

`create_publisher` déclare que le nœud publie des messages de type `Twist` (importé du module `geometry_msgs.msg`), sur le topic nommé `"/turtle1/cmd_vel"` de `TurtleSim`, et que la "taille de la file d'attente" est de 10. La taille de la file d'attente est un paramètre de QoS (qualité de service) requis qui limite le nombre de messages mis en file d'attente si un subscriber ne les reçoit pas assez rapidement.

Ensuite, un timer est créé avec un callback à exécuter toutes les 0,5 secondes :

```
def __init__(self):
    super().__init__('draw_circle')
    self.cmd_vel_pub = self.create_publisher(Twist, "/turtle1/cmd_vel", 10)
    timer_period = 0.5 # seconds
    self.timer = self.create_timer(timer_period, self.send_velocity_command)
    self.get_logger().info("Draw circle node has been started")
```

4. `send_velocity_command` crée un message du type `Twist` et assigne une valeur pour la vitesses linéaire `x` et la vitesse angulaire `z`. (Rappel : la tortue se trouvant dans un espace de déplacement 2D, elle ne peut bouger linéairement que sûr `x` et `y`, et angulairement uniquement sur `z`).

```
def send_velocity_command(self):
    msg = Twist()
    msg.linear.x = 2.0
    #msg.linear.y = 1.0
    msg.angular.z = 1.0
    self.cmd_vel_pub.publish(msg)
```

Ensuite, il y a la présence de la fonction qui publie le message `Twist`.

- Enfin, la fonction principale, `main()`, est définie :

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

La bibliothèque `rclpy` est d'abord initialisée, puis le nœud est créé avec la class `DrawCircleNode()`, et enfin le code fait "spin" le nœud pour que la fonction "send_velocity_command" soient appelées toutes les 0,5s.

7.4 Modifier les points d'entrée de votre package

- Rendez-vous dans le répertoire `~/ros2_ws/src/my_turtle_controller/`
- Ouvrez avec l'éditeur `nano`, le fichier "setup.py". Modifier ensuite la section entry-point comme suit :

```
entry_points={
    'console_scripts': [
        "draw_circle = my_turtle_controller.draw_circle:main"
    ],
},
```

7.5 Construire le package et sourcer l'overlay

- Retournez à la racine de votre espace de travail avec la commande `$ cd ~/ros2_ws` (Voir 2.2.4)
- Faites la vérification des dépendances avec les commandes suivantes (Explication dans la section 6.8.2) :


```
$ rosdep init
$ rosdep update
$ rosdep install -i --from-path src --rosdistro humble -y
```
- Construisez votre package en reconstruisant tous votre workspace en tapant la commande :

```
insa@ROS2:~/ros2_ws$ colcon build
```

- Votre package est construit et peut donc être utilisé avec ROS2. Pour ce faire nous devons mettre la source de l'overlay :

```
insa@ROS2:~/ros2_ws$ source install/setup.bash
```

- N'oubliez pas cette étape sinon la commande suivante ne fonctionnera pas car vous n'aurez pas sourcé votre "overlay".*

7.6 Tester votre package Python

- Souvenez vous de la structure de la commande permettant de lancer votre package :

```
$ ros2 run <package_name> <node_name>
```

Dans notre cas, notre package se nomme "my_turtle_controller". `<node_name>` correspond au nom du noeud de l'exécutable de notre package : "draw_circle".

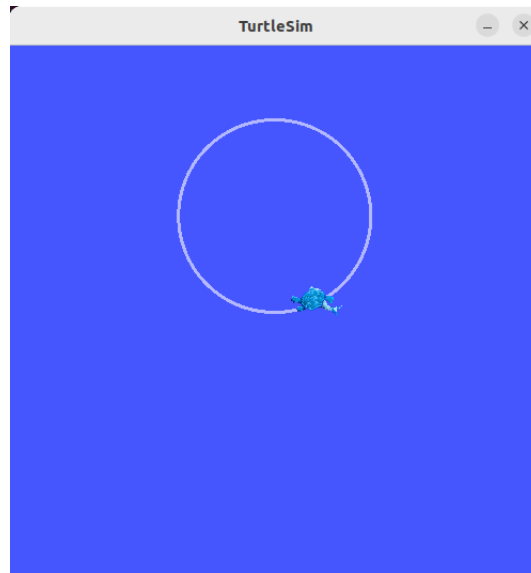
- Ouvrez TurtleSim avant de lancer votre package :

```
insa@ROS2:~$ ros2 run turtlesim turtlesim_node
```


3. Testez votre package en tapant la commande :

```
insa@ROS2:~/ros2_ws$ ros2 run my_turtle_controller draw_circle  
[INFO] [1697976143.477896420] [draw_circle]: Draw circle node has been started
```

Vous pourrez voir que la tortue commencera à bouger sous la forme d'un cercle dans TurtleSim :



8 Bibliographie

- [ROS2 Documentation : NOEUDS/TOPIC/SERVICES/ACTIONS](#)
- [ROS2 Documentation : Humble](#)
- Tutoriel ROS Noetic, *Guillaume Hansen et Alexandre Thouvenin*, 2023
- How to Control a Real TurtleBot with ROS through a Remote Raspberry Pi as ROS Master and with an OptiTrack Motion Capture System, *Sylvain Durand*, 2023
- [Robot Operating System](#), *Olivier Stasse*