
jupyter-ros

Release 0.6.0

Wolf Vollprecht

Oct 19, 2022

GENERAL

| | | |
|----------|----------------------------------|-----------|
| 1 | Introduction | 1 |
| 1.1 | Getting Started | 1 |
| 1.2 | Support | 1 |
| 1.3 | Installation | 2 |
| 1.4 | ROS Essentials | 2 |
| 1.5 | ROS 3D Widgets | 5 |
| 1.6 | ROS Server Extension | 7 |
| 1.7 | Troubleshooting | 8 |
| 1.8 | Developer Installation | 9 |
| 1.9 | Testing | 10 |
| 1.10 | Contributing | 11 |
| 1.11 | ROS Widgets | 13 |
| 1.12 | ROS2 Widgets | 18 |
| 2 | Indices and tables | 19 |
| | Python Module Index | 21 |
| | Index | 23 |

INTRODUCTION


jupyter-ros comes with many helpers to make working with ROS and rospy source code easier inside a Jupyter Notebook. The jupyter-ros tool suite contains the following pieces:

- publishing and receiving messages from the notebook interface
- a 3D widget for inspecting the robot and sensors in 3D
- playing back bag files and other helpers

1.1 Getting Started

The easiest way to get started is to create a ROS environment with conda or mamba with the following packages:

- jupyter-ros
- jupyterlab
- ros-noetic-desktop

```
$ conda create -n ros_env jupyter-ros jupyterlab ros-noetic-desktop -c conda-forge -c robostack
$ conda activate ros_env

# Launch JupyterLab
$ jupyter lab
```

At this point, you should be able to use jupyter-ros from any Jupyter notebook in the same conda environment.

```
import jupyros
```

For alternative methods of installing Jupyter-ROS, please visit [Installation](#) and [Developer Installation](#).

1.2 Support

If you stumble upon any issues or simply have general questions regarding Jupyter-ROS, please reach out. The fastest way to communicate with the core-team is through our Gitter channel [RoboStack](#). We monitor this channel regularly to help you get the most out of Jupyter-ROS.

You can also post issues on [GitHub](#). Please take a look at our contributing guidelines for information regarding what to include in a new issue. We are looking forward to your input!

1.3 Installation

Jupyter-ROS is distributed as a conda, pip, and npm package. We recommend installing it inside a conda environment. To install conda, we suggest using the [Miniconda installer](#).

```
# Option 1. conda or mamba [Recommended]
$ conda create -n jupyros_env python=3.9
$ conda activate jupyros_env
$ conda install jupyter-ros -c robostack
```

```
# Option 2. pip
$ pip install jupyros
```

```
# Option 3. npm
$ npm i jupyter-ros
```

1.3.1 The jupyter-ros server extension

The jupyter-ros package contains an optional server extension which can be used to serve static files from a catkin workspace to the web browser. For the 3D widgets we might have to load meshes to display robots correctly. These meshes are usually part of a robot description package in ROS. In order to enable the web browser to access those meshes, we can enable the jupyter-ros server extension:

```
$ jupyter serverextension enable jupyros

Enabling: jupyros
- Writing config: /home/user/.jupyter
  - Validating...
    jupyros 0.6.0 OK
```

The jupyter-ros server extension adds a handler to the Jupyter server that will return contents from ROS packages. To check that it works, you can (re-)start the Jupyter server and navigate to `localhost:8888/rospkg/rospy/package.xml` or any other file that should exist in your catkin workspace.

Warning: There are currently **no** security features in the jupyter-ros server extension. If you have sensitive ROS packages, all contents (including uncompiled source code) can be found through this extension.

1.4 ROS Essentials

1.4.1 Subscribing to a ROS topic

The Jupyter-ROS tools help publish and receive messages in a Jupyter notebook.

For publishing, the package contains tools to automatically generate widgets from message definitions. For receiving, the jupyter-ros package contains a helper that redirects output to a specific output widget (instead of spamming the entire notebook).

```
import jupyros as jr
import rospy
from std_msgs.msg import String

rospy.init_node('jupyter_node')
jr.subscribe('/sometopic', String, lambda msg: print(msg))
```

This creates a output widget, and buttons to toggle (stop or start) receiving messages. Internally, jupyter-ros will save a handle to the subscriber thread. Note that we did not use the rospy-way of creating a subscriber, but delegated this to the jupyter-ros package.

If we now send a message from a JupyterLab terminal, we see message being printed to the output widget below the cell where we executed the *jr.subscribe*.

```
$ rostopic pub /sometopic std_msgs/String "data: 'hello jupyter'" -r 10
```

Stop

```
data: "hello jupyter"
data: "hello jupyter"
data: "hello jupyter"
```

1.4.2 Publishing to a ROS topic

In the same way we can publish to a ROS topic by using the *jr.publish* helper.




```
import jupyros as jr
import rospy
from std_msgs.msg import String

rospy.init_node('jupyter_node')
jr.publish('/sometopic', String)
```

This results in a jupyter widget where one can insert the desired message in the text field. The form fields (jupyter widgets) are generated automatically from the message definition. If a different message type is used, different fields will be generated. For example, a `Vector3` message type contains three float fields (x,y, and z) for which we will get three `FloatTextField` instances; these can only hold float values (and not text).

```
from geometry_msgs.msg import Vector3

jr.publish('/vectortopic', Vector3)
```

| | | |
|---|---------------------------------|---|
| x | <input type="text" value="3"/> |  |
| y | <input type="text" value="7"/> |  |
| z | <input type="text" value="11"/> |  |

☐ Latch Message

1.4.3 Calling a ROS service

The same principles of publishing and receiving messages are applied for calling ROS services from a Jupyter notebook. Assuming that the service for adding two integers is available, a service client widget can be created with the following:

```
import jupyteros
import rospy
from rospy_tutorials.srv import AddTwoInts

rospy.init_node('service_node')
jupyteros.service_client('service_name', AddTwoInts)
```

The generated widget will change depending on the message type being passed.

| | |
|---|--------------------------------|
| a | <input type="text" value="8"/> |
| b | <input type="text" value="9"/> |
| <input type="button" value="Call Service"/> | |

1.4.4 Calling a ROS action server

A widget can also be created to call ROS action servers.

```
import jupyteros
import rospy
from actionlib_tutorials.msg import FibonacciAction, FibonacciGoal

rospy.init_node('action_node')
```

As an example, the Fibonacci server can be initialized from a JupyterLab terminal.

```
$ rosrund actionlib_tutorials fibonacci_server.py
```

The widget for the server called 'fibonacci' can then be generated as follows:

```
jupyteros.action_client('fibonacci', FibonacciAction, FibonacciGoal, callbacks={})
```

```
[INFO] [1658935103.937112]: [FIBONACCI] Waiting for action server.
[INFO] [1658935104.009992]: [FIBONACCI] Connection to server successful.
```

| | |
|---|---------------------------------|
| order | <input type="text" value="10"/> |
| <input type="button" value="Send Goal"/> <input type="button" value="Cancel Action"/> | |

The fields for the widget depend on the definition for the action *goal*.

1.4.5 Turtlesim

A widget for the most popular *turtlesim* animation is also included in Jupyter-ROS. The widget can be displayed with the code below, showing the default parameters.

```
import jupyteros
import rospy
from jupyteros import TurtleSim

turtlesim = TurtleSim(width=1600,
                      height=1600,
                      turtle_size=100,
                      background_color="#4556FF")

display(turtlesim.canvas)
```

When initialized, the widget will display a single turtle in the center of the canvas. The turtle images are randomized, so different turtles will appear after each run. Multiple turtles can also be spawned on the same canvas given a desired position (within the canvas limits) and orientation.

```
turtlesim.spawn(name="turtle2", pose={"x": 630,
                                     "y": 1260,
                                     "theta": math.radians(90)})
```

The turtles can be moved around by subscribing to topics such as */turtle1/pose* to receive new *Pose* messages.

```
# Retrieve current poses of the two turtles
new_poses = {"turtle1": turtlesim.turtles["turtle1"].pose,
             "turtle2": turtlesim.turtles["turtle2"].pose}

# Change the pose for the second turtle
new_poses["turtle2"] = {"x": 800, "y": 300, "theta": 2.5}

# Update the canvas
turtlesim.move_turtles(new_poses)
```

1.5 ROS 3D Widgets

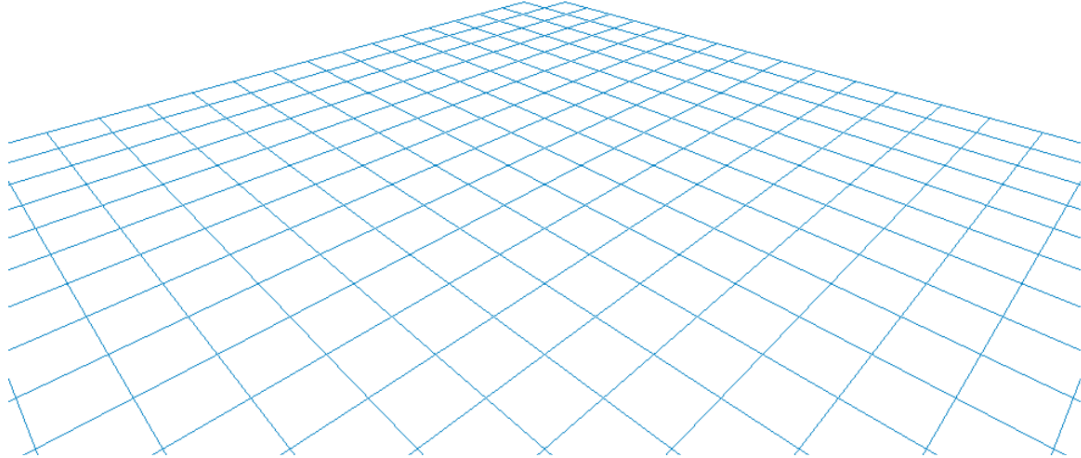
In order to visualize robots in the Jupyter notebook, jupyter-ros ships with the *ros3d* widgets. Jupyter widgets (*ipywidgets*) allow for communication between the Python “Kernel” and the JavaScript frontend.

For jupyter-ros, we have utilized the powerful *ROS3D.js* library to create widgets for the jupyter notebook frontend. That means that it’s easily possible to create some robot viewers inside the notebook!

```
from jupyteros import ros3d

v = ros3d.Viewer()
v.objects = [ros3d.GridModel()]
v
```

```
In [7]: from jupyteros import ros3d
In [8]: v = ros3d.Viewer()
In [9]: v.objects = [ros3d.GridModel()]
In [10]: v
```



ROS3D communicates with ROS via websocket. This communication is configured through the jupyter widgets protocol, but you are also required to run the “rosbridge websocket” package in your ROS environment (or launch file).

For this, you need to make sure that you have `ros-noetic-rosbridge-suite` and `ros-noetic-tf2-webrepublisher`.

Then you can run the following launch file to start up the necessary servers:

```
<launch>
  <include file="$(find rosbridge_server)/launch/rosbridge_websocket.launch" />
  <node pkg="tf2_web_republisher" type="tf2_web_republisher" name="tf2_web_republisher" /
  </node>
</launch>
```

Warning: The currently shipped rosbridge suite uses a very old version of the Tornado web server. This version is *incompatible* with the more recent version used by Jupyter. If you run both (Jupyter and rosbridge) from the same Python environment, it is likely to silently fail (no connection from the ROS 3D widgets to the rosbridge).

Therefore we recommend to separate the two environments by using conda, and a generated ROS kernel.

Once the rosbridge websocket is running, we can configure a ROS connection in the Jupyter notebook, and subscribe to a topic. For example, in order to subscribe to a laser scan we can use the following code:

```
from jupyteros import ros3d

v = ros3d.Viewer()
rc = ros3d.ROSConnection(url="ws://localhost:9090")
tf_client = ros3d.TFClient(ros=rc, fixed_frame='')

laser_view = ros3d.LaserScan(topic="/scan", ros=rc, tf_client=tf_client)
g = ros3d.GridModel()
v.objects = [g, laser_view]
```

(continues on next page)

(continued from previous page)

v

This will now attempt to connect to the websocket at `ws://localhost:9090` (which should be the address given to the rosbriidge). And if successful, it should receive data from `/scan` topic and display it in the 3D viewer.

Note that, similar to RViz, you can select a fixed frame, which will appear at the origin of the viewer.

Besides the laser scan view, you can add many other visualizations to the viewer. ROS3D contains classes for a Robot (URDFModel), occupancy grid (OccupancyGrid), regular and interactive markers (Marker and InteractiveMarker), Pose, Polygon, PoseArray, Path, and PointCloud.

If you want to visually change how your Viewer looks, you can use the layout attribute (this works the same way [across many of the ipywidgets](#)).

```
from jupyros import ros3d
from ipywidgets import Layout

v = ros3d.Viewer()
v.layout = Layout(border="3px solid red", width="200px", height="50px")
# The following would have had the same effect:
# v.layout.border = "3px solid red" ...
```

```
In [5]: from jupyros import ros3d
        from ipywidgets import Layout

        v = ros3d.Viewer()
        v.layout = Layout(border="3px solid red", width="200px", height="100px")
        # The following would have had the same effect:

In [6]: v
```



1.6 ROS Server Extension

The jupyter-ros package comes with a jupyter server extension to serve static files (such as robot models) from a catkin workspace.

Once the server extension is installed, you can point the URDFModel URL parameter to `http://localhost:8888/rospkg/` in order for it to search below that URL for the meshes and other required assets.

The endpoint will use `rospkg` to find assets in your catkin workspace. For example, the URL `http://localhost:8888/rospkg/roscpp/CMakeLists.txt` should return the CMakeLists file of the roscpp package. This feature is mainly useful for sending mesh files to the frontend.

Warning: Currently, there is **no** mechanism in place to filter requests based on file type. That means, all your source files can be accessed through this endpoint.

1.7 Troubleshooting

When working with the official ROS packages, it may be necessary to modify the Jupyter kernel.

Once you have Jupyter and the *jupyteros* package installed, you can run the kernel generator to install a special ROS Python kernel for Jupyter. This modifies a pre-existing kernel in such a way that it knows about the catkin workspace.

You can find the available jupyter kernels by running:

```
$ jupyter kernelspec list

Available kernels:
julia          ~/.local/share/jupyter/kernels/julia
python3        ~/miniconda3/share/jupyter/kernels/python3
python2        ~/miniconda2/share/jupyter/kernels/python2
xcpp11         ~/miniconda3/share/jupyter/kernels/xcpp11
xcpp14         ~/miniconda3/share/jupyter/kernels/xcpp14
xcpp17         ~/miniconda3/share/jupyter/kernels/xcpp17
xonsh          ~/miniconda3/share/jupyter/kernels/xonsh
```

For maximum compatibility with ROS 1 releases, we want to base the ROS kernel on the existing Python 2 kernel.

Run the installed generator:

```
$ ros_kernel_generator python2 /home/$USER/catkin_ws/devel/setup.bash
```

This will install a new kernel specification next to the installed python2 kernel (in this case at `~/miniconda2/share/jupyter/kernels/ros_python2`) with ROS specific environment variables set to the ones from the catkin workspace (such as an additional Python path to find the ROS Python libraries).

1.7.1 Websocket Failure

For troubleshooting the ROS 3D widgets, we have prepared the following tips.

The 3D widgets of ROS communicate with the backend through websockets. To make sure that a websocket connection is established, first, open the “Inspector” (in FireFox or Chrome, right click -> Inspect Element) and navigate to the “Network” panel of the Inspector. The network panel shows all requests. If the notebook cells are now executed we should see a “Websocket” connection being established.

If the websocket connection remains at “Pending...” then you might be using the wrong Tornado version for the rosbridge websocket server. Using the wrong Tornado version results in a **silent failure**. In this case, it is very important to **not** mix the Jupyter Python environment with the rosbridge environment as Jupyter uses a more recent Tornado version than the default rosbridge websocket installed from the official ROS packages.

Therefore, the fix is usually to reinstall the tornado version from the APT packages:

```
sudo apt-get install python-tornado
sudo apt-get install ros-melodic-rosbridge-suite --reinstall
```

You can also check the network panel to make sure that mesh files are loaded correctly.

1.8 Developer Installation

1.8.1 Install Jupyter-ROS

1. Clone repository

```
$ git clone git@github.com:RoboStack/jupyter-ros.git
$ cd jupyter-ros
```

2. Create a conda environment for development with the following packages

- python = 3.9
- jupyterlab
- jupyter-packaging
- nodejs <= 15
- ros-noetic-desktop

```
# You can use conda as well
$ mamba create -n jupyros_env python=3.9 jupyterlab jupyter-packaging nodejs=15 ros-
noetic-desktop -c conda-forge -c robostack

$ mamba activate jupyros_end
```

3. Install *jupyter-ros* in editable mode

```
# From the jupyter-ros root directory
$ pip install -e .
```

4. Symlink the JupyterLab extension

```
$ jupyter labextension develop . --overwrite
```

5. Verify installation with Python

```
import jupyros
print(jupyros.__file__)
# Should return /home/user/jupyter-ros/jupyros/__init__.py
```

1.8.2 Build Documentation

1. Create a new conda environment with the following dependencies:

- sphinx
- myst-parser
- jinja2 <= 3.0
- sphinx-rtd-theme

```
# You can use conda as well
$ mamba create -n jupyros_docs sphinx myst-parser jinja2=3.0 -c conda-forge
$ mamba activate jupyros_docs
$ pip install sphinx-rtd-theme
```

2. **[Optional]** Install `jupyter-ros` in the environment. This is only necessary for the *References* page to display correctly; otherwise, there will be a few warnings in the next step.

3. Build the documents

```
$ cd jupyter-ros/docs/
$ make html
```

4. Open the documentation locally

```
$ cd build/html/
$ python -m http.server
```

5. From a web browser, navigate to `localhost:8000`

1.9 Testing

The simplest way to test any additions to Jupyter-ROS is to create a fresh environment and install the package in development mode.

1. Create new test environment with minimal packages

```
$ conda create -n test_env python=3.9 jupyterlab nodejs=15 jupyter-packaging ros-
↪noetic-desktop -c conda-forge -c robostack

$ conda activate test_env
```

2. Navigate to the root directory and install `jupyter-ros`

```
$ cd jupyter-ros/
$ pip install -e .
```

If there are any errors with this step, this indicates that the new additions are not configured correctly for installation. This will require some additional troubleshooting, but a common issue is to forget to include newly required dependencies in the `setup.py` file.

```
setup_args = {
    'install_requires': [
        'ipywidgets>=7.0.0',
        'bqplot',
        'numpy',
        'rospkg',
        'ipycanvas'
    ],
}
```

3. Symlink the JupyterLab extension and verify that `jupyros` can be imported.

```
$ jupyter labextension develop . --overwrite
$ jupyter lab
```

```
import jupyros
```

This step may also require some troubleshooting depending on the changes made to Jupyter-ROS. If you see interference from other conda environments, e.g. additional lab extensions which should not be enabled in the test environment, it is often helpful to remove the `~/ .jupyter` directory.

```
# List all the lab extensions
$ jupyter labextension list

# Remove jupyter directory [optional]
$ rm -r ~/ .jupyter
```

4. Once the setup is complete, it is now time to test the new additions. This type of testing will vary greatly depending on the changes, we suggest to use your best judgement.

1.10 Contributing

First off, thank you for considering contributing to Jupyter-ROS . It's people like you that make Jupyter-ROS such a great tool.

Following these guidelines helps to communicate that you respect the time of the developers managing and developing this open source project. In return, they should reciprocate that respect in addressing your issue, assessing changes, and helping you finalize your pull requests.

Jupyter-ROS is an open source project and we love to receive contributions from our community — you! There are many ways to contribute. For example, you can

- write a new tutorial or a blog post
- improve the documentation or the existing examples
- submit bug reports or feature requests
- write code to incorporate into Jupyter-ROS itself

1.10.1 Ground Rules

We welcome all kinds of contribution and value them highly. We pledge to treat everyone's contribution fairly and with respect, and we are here to bring awesome pull requests over the finish line.

Please note that we adhere to the [Python Community Code of Conduct](#) and by contributing to this project you also agree to follow the same guidelines.

1.10.2 Your First Contribution

Working on your first Pull Request? Here are some resources to help you get started:

- [First Timers Only](#)
- [Make a Pull Request](#)
- [How to Contribute to an Open Source Project on GitHub](#)

At this point, you're ready to make your changes! Feel free to ask for help; everyone is a beginner at first .

If a maintainer asks you to “rebase” your PR, they're saying that a lot of code has changed and that you need to update your branch so that it's easier to merge.

1.10.3 Getting Started

1. Create your own fork of the code.
2. Do the changes in your fork.
3. If your changes only involve spelling or grammar fixes, move to step 7.
4. Test your changes in a clean environment and update installation instructions and dependencies as needed.
5. When adding new features, make sure to update the documentation and provide an example under *notebooks/*.
6. New notebooks
 - Remove all output.
 - Remove unnecessary cells.
 - Include a descriptive title.
 - Specify ROS version in the notebook name, “*ROS Turtlesim.ipynb*” vs “*ROS2 Turtlesim.ipynb*”
 - Any additional steps the user needs to take to run all the cells in the notebook should be clearly stated in markdown cells.
7. If you are happy with your changes, create a pull request.

1.10.4 How to Report a Bug

Security

If you find a security vulnerability, do NOT open an issue. Email w.vollprecht@gmail.com instead. In order to determine whether you are dealing with a security issue, ask yourself these two questions:

- Can I access something that's not mine, or something I shouldn't have access to?
- Can I disable something for other people?

If the answer to either of those two questions are “yes”, then you're probably dealing with a security issue. Note that even if you answer “no” to both questions, you may still be dealing with a security issue, so if you're unsure, just email us.

Other bugs

When filing an issue, make sure to answer these five questions:

1. What version of *jupyteros* are you using?
2. What operating system and processor architecture are you using?
3. What did you do?
4. What did you expect to see?
5. What did you see instead?

General questions should be handled through [Gitter](#) instead of the issue tracker. The maintainers there will answer or ask you to file an issue if you've tripped over a bug.

1.10.5 How to Suggest a Feature or Enhancement

If you find yourself wishing for a feature that doesn't exist in Jupyter-ROS, you are probably not alone. There are bound to be others out there with similar needs. Many of the features that Jupyter-ROS has today have been added because our users saw the need. Open an issue on our [issues list](#) on GitHub which includes the following:

1. A description of the feature you would like to see
2. Why do you need it?
3. How should it work?

1.10.6 Code Review Process

Any change to resources in this repository must be through pull requests. This applies to all changes to documentation, code, binary files, etc. No pull request can be merged without being reviewed.

The core team looks at Pull Requests on a regular basis and provides feedback after each review. Once feedback has been given, we expect responses within three weeks. After the three weeks have elapsed, we may close the pull request if it isn't showing any activity.

A pull request will be merged once all the feedback has been addressed and there are no objections by any of the committers.

1.10.7 Community

You can chat with the core team on gitter.im/RoboStack. We try to answer all questions within 48 hours.

1.11 ROS Widgets

`jupyteros.ros1.pubsub.subscribe(topic, msg_type, callback)`

Subscribes to a specific topic in another thread, but redirects output!

@param topic The topic @param msg_type The message type @param callback The callback

@return Jupyter output widget

`jupyteros.ros1.ros_widgets.action_client(action_name, action_msg, goal_msg, callbacks=None)`

Create a form widget for message type `action_msg`. This function analyzes the fields of `action_msg` and creates an appropriate widget. An action client is automatically created which sends a goal to the action server given as `action_name`. This allows pressing the “Send Goal” button to send the message to ROS.

@param `action_name` The namespace in which to access the action @param `action_msg` The action message type @param `goal_msg` The goal message type @param `callbacks` A dictionary of callback function handles with

optional keys: `done_cb`, `active_cb` and `feedback_cb`

@return jupyter widget for display

`jupyteros.ros1.ros_widgets.add_widgets(msg_instance, widget_dict, widget_list, prefix='')`

Adds widgets.

@param `msg_type` The message type @param `widget_dict` The form list @param `widget_list` The widget list

@return `widget_dict` and `widget_list`

`jupyteros.ros1.ros_widgets.bag_player(bagfile='')`

Create a form widget for playing ROS bags. This function takes the bag file path, extracts the bag summary and play the bag with the given arguments.

@param `bagfile` The ROS bag file path

@return jupyter widget for display

`jupyteros.ros1.ros_widgets.client(*args, **kwargs)`

Deprecated client for ROS services. Use `service_client()` instead.

`jupyteros.ros1.ros_widgets.publish(topic, msg_type)`

Create a form widget for message type `msg_type`. This function analyzes the fields of `msg_type` and creates an appropriate widget. A publisher is automatically created which publishes to the topic given as topic parameter. This allows pressing the “Send Message” button to send the message to ROS.

@param `msg_type` The message type @param `topic` The topic name to publish to

@return jupyter widget for display

`jupyteros.ros1.ros_widgets.service_client(srv_name, srv_type)`

Create a form widget for message type `srv_type`. This function analyzes the fields of `srv_type` and creates an appropriate widget.

@param `srv_type` The service message type @param `srv_name` The service name to call

@return jupyter widget for display

`class jupyteros.ros1.ros3d.DepthCloud(**kwargs: Any)`

Display a Depth Cloud for a RGB-D cloud (needs infrastructure on the server side.)

`class jupyteros.ros1.ros3d.GridModel(**kwargs: Any)`

A simple GridModel

Displays a 3D grid.

Parameters

- **cell_size** (*Float*) – Size of the cells in meters
- **color** (*Unicode*) – Color of the grid lines (e.g. a hex specifier `#FF0000`)
- **num_cells** (*Int*) – Number of cells in length, and width (default: 2)0

class jupyteros.ros1.ros3d.**InteractiveMarker**(**kwargs: Any)

Interactive Marker Widget

Displays an interactive marker in the Viewer.

Parameters

- **ros** (*ROSConnection instance*) – Instance of the ROS Connection that should be used
- **tf_client** (*TFClient instance*) – Instance of the TF Client that should be used
- **topic** (*Unicode*) – The topic to subscribe to (default: `/basic_controls`)
- **menu_font_size** (*Unicode*) – Menu font size (default: `'0.8em'`)

class jupyteros.ros1.ros3d.**LaserScan**(**kwargs: Any)

Displays a LaserScan message

Parameters

- **ros** (*ROSConnection instance*) – Instance of the ROS Connection that should be used
- **tf_client** (*TFClient instance*) – Instance of the TF Client that should be used
- **topic** (*Unicode*) – Name of the topic (default: `/scan`)
- **point_ratio** (*Float*) – Ratio of points to send to the frontend (default: 1.0)
- **message_ratio** (*Float*) – Ratio of messages to send to the frontend (default: 1.0)
- **max_points** (*Int*) – Maximum number of points to display (default: 200000)
- **color_source** (*Unicode*) – Source field for the color information (default: `'intensities'`)
- **color_map** (*Unicode*) – Color map function (default: `''`)
- **point_size** (*Float*) – Point size (default: 0.05)
- **static_color** (*Unicode*) – Hex string of the color for visualization (default: `"#FF0000"`)

class jupyteros.ros1.ros3d.**Marker**(**kwargs: Any)

Displays a Marker message

Parameters

- **ros** (*ROSConnection instance*) – Instance of the ROS Connection that should be used
- **tf_client** (*TFClient instance*) – Instance of the TF Client that should be used
- **topic** (*Unicode*) – Name of the topic (default: `/pose_array`)
- **path** (*Unicode*) – Marker path (default: `/`)
- **lifetime** (*Float*) – Lifetime of marker in seconds, 0 for infinity (default: 0.0)

class jupyteros.ros1.ros3d.**MarkerArrayClient**(**kwargs: Any)

A client that listens to changes in a MarkerArray and triggers the update of a visualization.

Parameters

- **ros** (*ROSConnection instance*) – Instance of the ROS Connection that should be used
- **tf_client** (*TFClient instance*) – Instance of the TF Client that should be used
- **topic** (*Unicode*) – Name of the topic (default: `/marker_array`)
- **path** (*Unicode*) – The base path to any meshes that will be loaded (default: `'/'`)

class jupyteros.ros1.ros3d.OccupancyGrid(**kwargs: Any)

Displays an Occupancy Grid

Parameters

- **ros** (*ROSConnection instance*) – Instance of the ROS Connection that should be used
- **tf_client** (*TFClient instance*) – Instance of the TF Client that should be used
- **topic** (*Unicode*) – The topic to subscribe to (default: /basic_controls)
- **continuous** (*Bool*) – Whether the occupancy grid should continuously update (default: False)
- **compression** (*Unicode*) – Compression mechanism (default: 'cbor')
- **color** (*Unicode*) – Color of the grid lines (e.g. a hex specifier #FFFFFF)
- **opacity** (*Float*) – Opacity of the occupancy grid (default: 1.0)

class jupyteros.ros1.ros3d.Path(**kwargs: Any)

Displays a Path message

Parameters

- **ros** (*ROSConnection instance*) – Instance of the ROS Connection that should be used
- **tf_client** (*TFClient instance*) – Instance of the TF Client that should be used
- **topic** (*Unicode*) – Name of the topic (default: /path)
- **color** (*Unicode*) – Hex string of the color for visualization (default: "#CC00FF")

class jupyteros.ros1.ros3d.PointCloud(**kwargs: Any)

Displays a PointCloud message

Parameters

- **ros** (*ROSConnection instance*) – Instance of the ROS Connection that should be used
- **tf_client** (*TFClient instance*) – Instance of the TF Client that should be used
- **topic** (*Unicode*) – Name of the topic (default: /point_cloud)
- **point_ratio** (*Float*) – Ratio of points to send to the frontend (default: 1.0)
- **message_ratio** (*Float*) – Ratio of messages to send to the frontend (default: 1.0)
- **max_points** (*Int*) – Maximum number of points to display (default: 200000)
- **point_size** (*Float*) – Point size (default: 0.05)
- **static_color** (*Unicode*) – Hex string of the color for visualization (default: "#FF0000")

class jupyteros.ros1.ros3d.Polygon(**kwargs: Any)

Displays a Polygon message

Parameters

- **ros** (*ROSConnection instance*) – Instance of the ROS Connection that should be used
- **tf_client** (*TFClient instance*) – Instance of the TF Client that should be used
- **topic** (*Unicode*) – Name of the topic (default: /polygon)
- **color** (*Unicode*) – Hex string of the color for visualization (default: "#CC00FF")

class jupyteros.ros1.ros3d.**Pose**(**kwargs: Any)

Displays a Pose message

Parameters

- **ros** (*ROSConnection instance*) – Instance of the ROS Connection that should be used
- **tf_client** (*TFClient instance*) – Instance of the TF Client that should be used
- **topic** (*Unicode*) – Name of the topic (default: /pose)
- **color** (*Unicode*) – Hex string of the color for visualization (default: "#CC00FF")
- **length** (*Float*) – Length of pose arrows (default: 1m)

class jupyteros.ros1.ros3d.**PoseArray**(**kwargs: Any)

Displays a PoseArray message

Parameters

- **ros** (*ROSConnection instance*) – Instance of the ROS Connection that should be used
- **tf_client** (*TFClient instance*) – Instance of the TF Client that should be used
- **topic** (*Unicode*) – Name of the topic (default: /pose_array)
- **color** (*Unicode*) – Hex string of the color for visualization (default: "#CC00FF")
- **length** (*Float*) – Length of pose arrows (default: 1m)

class jupyteros.ros1.ros3d.**ROSConnection**(**kwargs: Any)

Base ROS Connection

The ROS Connection widget has the parameters with the websocket endpoint to communicate with the rosbridge server.

Parameters

url (*Unicode*) – URL endpoint of the websocket. Defaults to `ws://{hostname}:9090` where {hostname} is replaced by the current hostname at runtime (e.g. localhost). You can override the default value by setting the JUPYTER_WEBSOCKET_URL environment variable.

class jupyteros.ros1.ros3d.**SceneNode**(**kwargs: Any)

Scene Node (to be used in conjunction with DepthCloud)

class jupyteros.ros1.ros3d.**TFClient**(**kwargs: Any)

Base TF Client

The TFClient keeps track of TF frames.

Parameters

- **angular_threshold** (*Float*) – The angular threshold for the TF republisher (default: 0.01)
- **translational_threshold** (*Float*) – The translation threshold for the TF republisher (default: 0.01)
- **rate** (*Float*) – Update and publish rate for the TF republisher (default: 10.0)
- **fixed_frame** (*Unicode*) – Fixed base frame for TF tree (default: “")

class jupyteros.ros1.ros3d.**URDFModel**(**kwargs: Any)

A URDFModel (Robot model)

Displays a 3D robot model.

Parameters

- **ros** (*ROSConnection instance*) – Instance of the ROS Connection that should be used
- **tf_client** (*TFClient instance*) – Instance of the TF Client that should be used
- **url** (*Unicode*) – URL from which to fetch the `_assets_` (mesh and texture files). This can be either the jupyter-ros server extension (in this case one should use “<http://{hostname}:{port}/rospkg/>”) or another server / URL where the mesh files can be downloaded (default: “<http://{hostname}:3000/>”)

class jupyteros.ros1.ros3d.**Viewer**(***kwargs: Any*)

Viewer class

This is the class that represents the actual 3D viewer widget. The viewer is derived from the `ipywidgets.DOMWidget` and therefore also implements the `layout` attribute which can be used to modify the CSS layout of the viewer.

Parameters

- **background_color** (*Unicode*) – Background color of the viewer (default: `'#FFFFFF'`)
- **alpha** (*Float*) – The alpha value of the background
- **objects** (*List of ROS3D widget instances*) – Objects to render in the viewer (e.g. `Marker`, `PointCloud`, ...)

1.12 ROS2 Widgets

Coming soon!

INDICES AND TABLES

- `genindex`
- `modindex`
- `search`

PYTHON MODULE INDEX

j

jupyros.ros1.pubsub, [13](#)
jupyros.ros1.ros3d, [14](#)
jupyros.ros1.ros_widgets, [13](#)
jupyros.ros1.turtle_sim, [18](#)

A

`action_client()` (in module `jupyteros.ros1.ros_widgets`), 13

`add_widgets()` (in module `jupyteros.ros1.ros_widgets`), 14

B

`bag_player()` (in module `jupyteros.ros1.ros_widgets`), 14

C

`client()` (in module `jupyteros.ros1.ros_widgets`), 14

D

`DepthCloud` (class in `jupyteros.ros1.ros3d`), 14

G

`GridModel` (class in `jupyteros.ros1.ros3d`), 14

I

`InteractiveMarker` (class in `jupyteros.ros1.ros3d`), 14

J

`jupyteros.ros1.pubsub`
module, 13

`jupyteros.ros1.ros3d`
module, 14

`jupyteros.ros1.ros_widgets`
module, 13

`jupyteros.ros1.turtle_sim`
module, 18

L

`LaserScan` (class in `jupyteros.ros1.ros3d`), 15

M

`Marker` (class in `jupyteros.ros1.ros3d`), 15

`MarkerArrayClient` (class in `jupyteros.ros1.ros3d`), 15

module

`jupyteros.ros1.pubsub`, 13

`jupyteros.ros1.ros3d`, 14

`jupyteros.ros1.ros_widgets`, 13

`jupyteros.ros1.turtle_sim`, 18

O

`OccupancyGrid` (class in `jupyteros.ros1.ros3d`), 15

P

`Path` (class in `jupyteros.ros1.ros3d`), 16

`PointCloud` (class in `jupyteros.ros1.ros3d`), 16

`Polygon` (class in `jupyteros.ros1.ros3d`), 16

`Pose` (class in `jupyteros.ros1.ros3d`), 16

`PoseArray` (class in `jupyteros.ros1.ros3d`), 17

`publish()` (in module `jupyteros.ros1.ros_widgets`), 14

R

`ROSConnection` (class in `jupyteros.ros1.ros3d`), 17

S

`SceneNode` (class in `jupyteros.ros1.ros3d`), 17

`service_client()` (in module `jupyteros.ros1.ros_widgets`), 14

`subscribe()` (in module `jupyteros.ros1.pubsub`), 13

T

`TFClient` (class in `jupyteros.ros1.ros3d`), 17

U

`URDFModel` (class in `jupyteros.ros1.ros3d`), 17

V

`Viewer` (class in `jupyteros.ros1.ros3d`), 18