### BlueOS

# Installation

Now with a stable release! Be aware that if you have a B.E.T.A before 1.0.0 release a fresh SD card flash is necessary.

## Download

BlueOS is a ground-up rewrite software to replace Companion. To use it you'll need to download and flash an SD card. It is compatible with both Raspberry Pi 3 and Raspberry Pi 4.

The latest available stable version is: [ [](https://github.com/bluerobotics/blueos/releases/latest/download/BlueOS-raspberry.zip)](https://github.com/bluerobotics/blueos/releases/latest/download/BlueOS-raspberry.zip)

## Flash

We recommend using a fresh SD card, with at least 4GB capacity.

1. Download and install [Balena Etcher](https://www.balena.io/etcher/)
2. Insert the SD card to your computer (you may need an SD card reader)
3. Open Etcher, select the image you just downloaded, and flash it onto the SD card

## Run

1. Eject your SD card with the new BlueOS software
2. Insert it into your Raspberry Pi, and power it up!

The first boot may take a couple of minutes, as it expands the filesystem to the new SD card capacity. It should take around 2 minutes for a 16GB class 10 SD card.

## Updates

Once BlueOS is installed, updating to a different version is simple via the [Version Chooser](https://docs.bluerobotics.com/ardusub-zola/software/onboard/BlueOS-1.0/advanced-usage/" \l "version-chooser).

## Web Interface

BlueOS is designed to be a modular collection of services, which generally each provide a REST API, documentation, and a development webpage. The web interface monitors the autopilot and other main software components, and also listens for and displays connections from other HTTP servers (on TCP ports) and docker containers, so you can keep your own integrations isolated from the normal BlueOS release/update cycle.

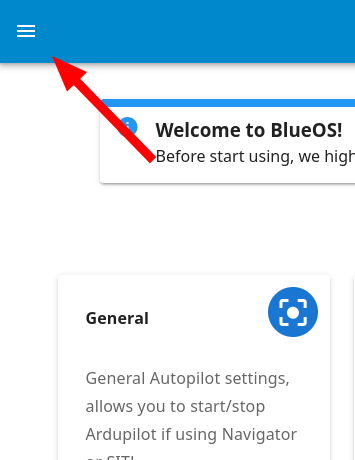
### Interface Access

You can access BlueOS via the old IP address ([192.168.2.2](http://192.168.2.2/)) or via [blueos.local](http://blueos.local/), to connect with it via wifi, it's possible to use [blueos-wifi.local](http://blueos-wifi.local/).

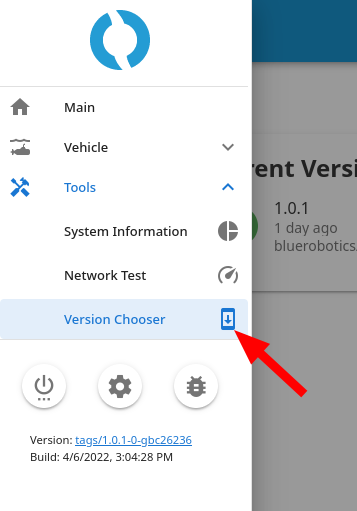
### Updating BlueOS

Now that your BlueOS has an internet connection, you can perform the update to the latest available version.

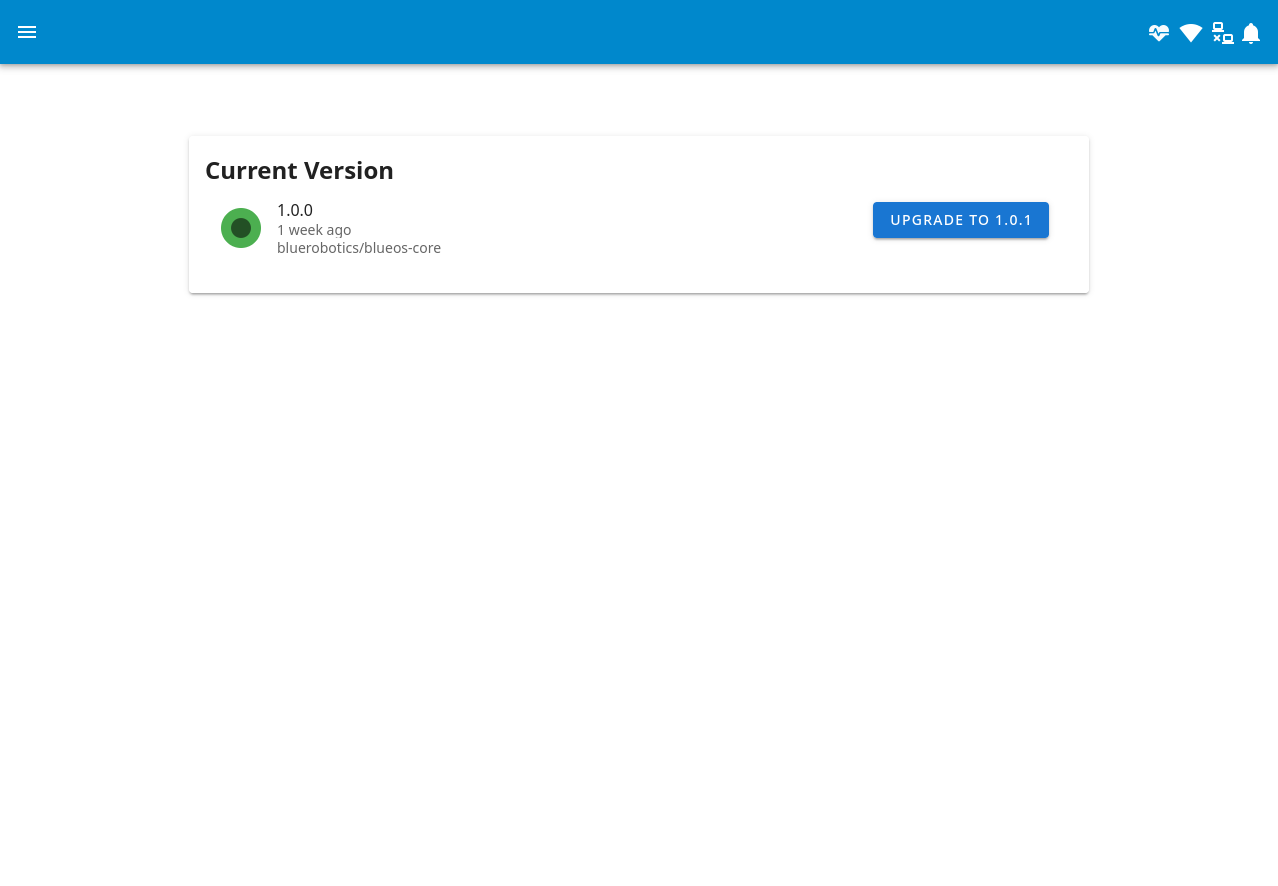
1. Click on the hamburger menu



1. Under tools, select **[Version-Chooser](https://docs.bluerobotics.com/ardusub-zola/software/onboard/BlueOS-1.0/advanced-usage/" \l "version-chooser)**



1. If you're already on the latest version, the right side of your Local Version will be blank. If not, you should see a blue Update button.



1. Once the update button is clicked the update process will run. Please wait until it finishes - it will automatically reload the webpage for you.

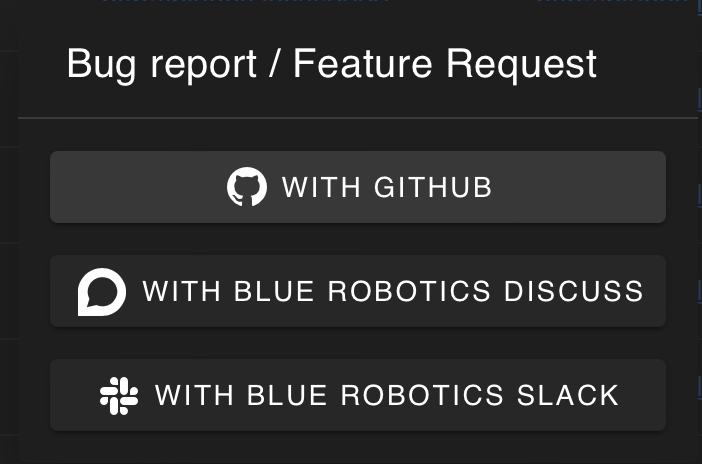
### Pirate Mode

The default BlueOS interface is simplified, and shows only the major tools that most people are likely to find useful. Full functionality is available via "Pirate Mode", which can be enabled from the [settings in the sidebar](https://docs.bluerobotics.com/ardusub-zola/software/onboard/BlueOS-1.0/advanced-usage/" \l "settings). Note that Pirate Mode is advanced/development mode, and should be used with care.

This documentation by default shows the full functionality interface, to provide an overview of all functionality instead of a limited subset, but if you're only interested in the basic functionality you can click the button below:

Any pages that are extended by (or only available in) Pirate Mode are shown in [dark mode](https://docs.bluerobotics.com/ardusub-zola/software/onboard/BlueOS-1.0/advanced-usage/" \l "settings), and described with grey text.

##### Feedback



Submit feedback about BlueOS via:

* [Issues on the GitHub repository](https://github.com/bluerobotics/BlueOS-docker/issues) (allows easily tracking changes, and notification when complete/fixed)
* Posts on [the Blue Robotics forum](https://discuss.bluerobotics.com/) (allows easy discussion with the community)

### General

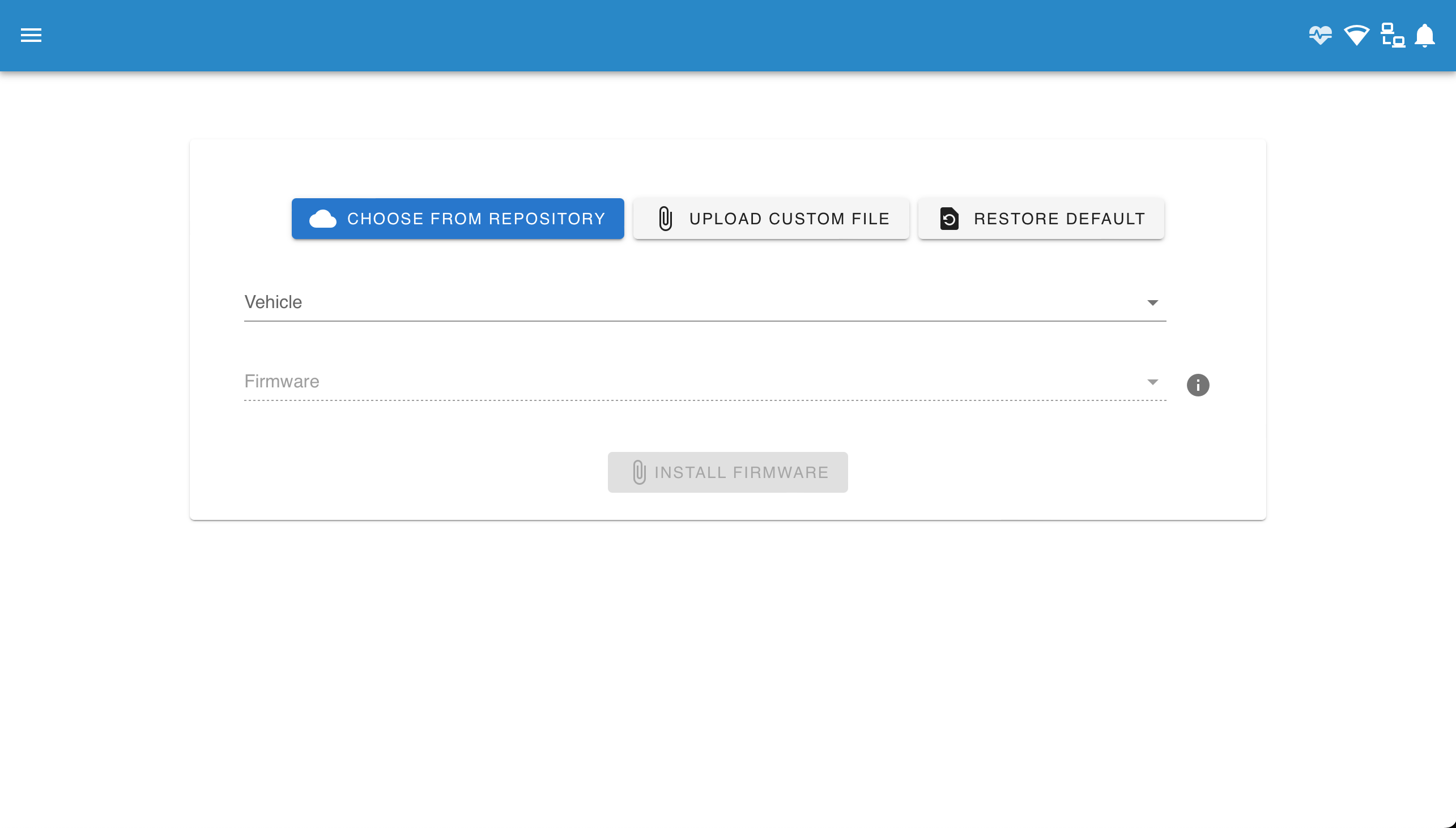
The "General" page provides basic info about the active autopilot, along with options to:

* Change board (select a connected board, or run an [SITL](https://www.ardusub.com/developers/sitl.html) simulation)
* Restart the autopilot
* Start the autopilot
* Stop the autopilot

### Firmware

Based On: [*ArduPilot Manager*](https://github.com/bluerobotics/BlueOS-docker/tree/1.0.1/core/services/ardupilot_manager) | Port:8000

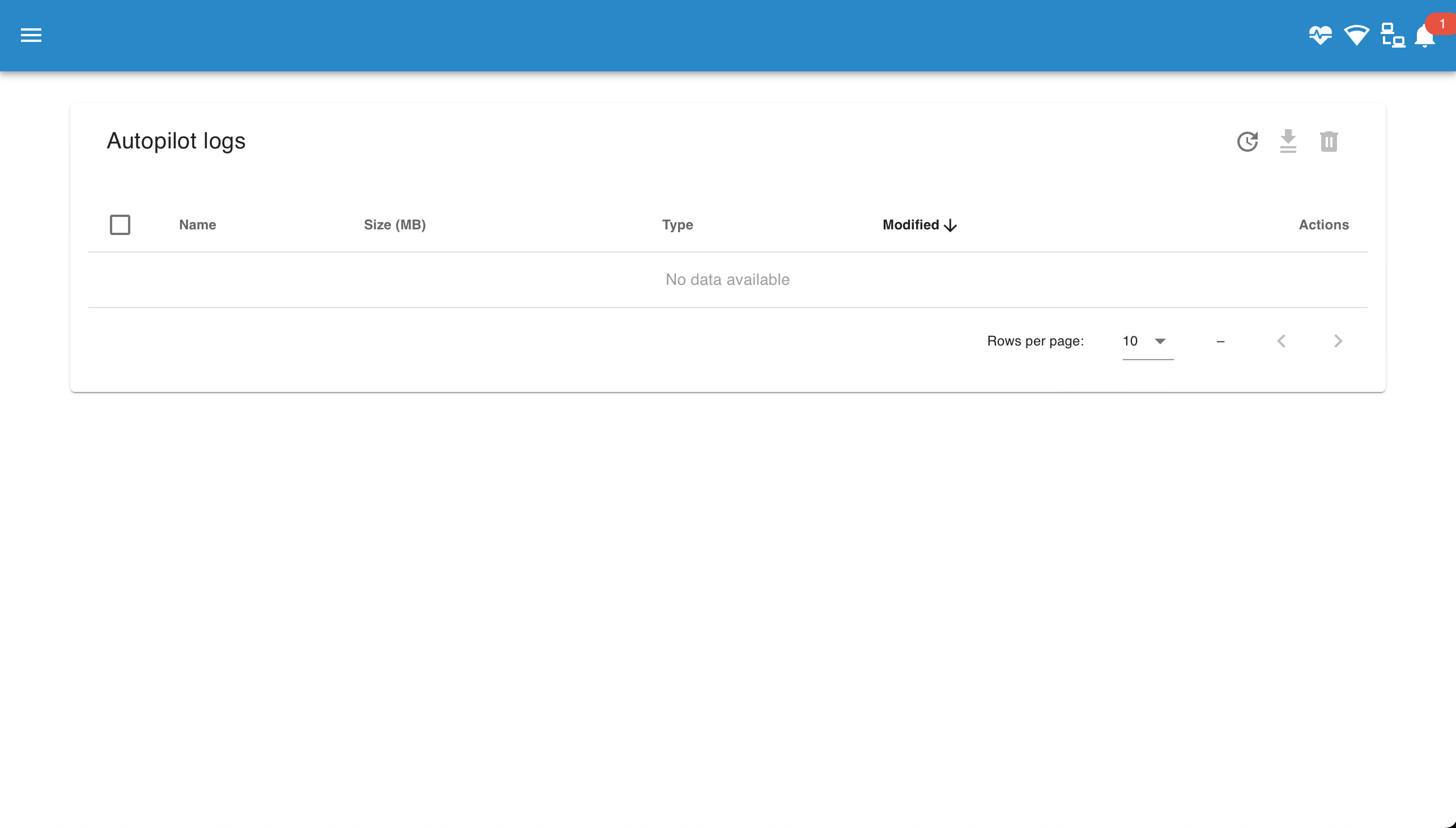
* [ArduPilot](https://docs.bluerobotics.com/ardusub-zola/software/onboard/BlueOS-1.0/advanced-usage/ardupilot.org) family of firmwares only
* Choose firmware to install
  + Select from the online repository
    - Select vehicle type (Sub / Rover / Plane / Copter)
    - Select desired release and stability level
      * Official - The latest stable release. Recommended for most users.
      * Stable - A production-ready release. Suitable for most users.
        + e.g. Stable-4.0.3
      * Beta - In-testing release, with new features and improvements, aiming to become stable. May have bugs.
      * Dev - Development branch, with all the newest features. Intentionally unstable (changes quickly), and possible untested/dangerous.
  + Upload a custom firmware file from the surface computer
  + Restore the default (ArduSub) firmware for the connected flight controller
* Flash firmware onto a connected compatible flight controller board



### Log Browser

Based On: [*UAV LogViewer*](https://ardupilot.org/copter/docs/common-uavlogviewer.html)

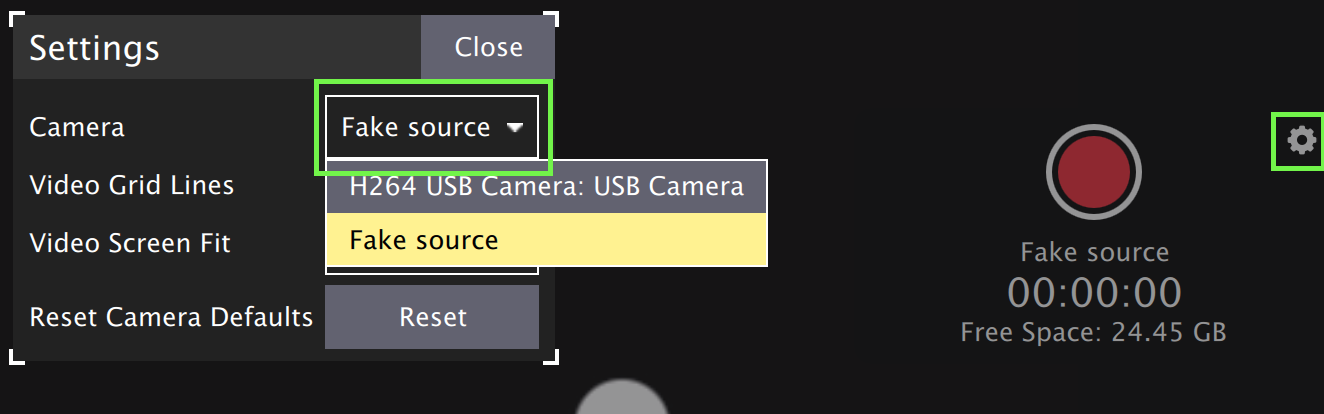
* Built in log viewer for powerful analysis of vehicle telemetry
* Currently only set up to fetch logs automatically from Linux-based autopilots
  + e.g. Pixhawk not yet supported



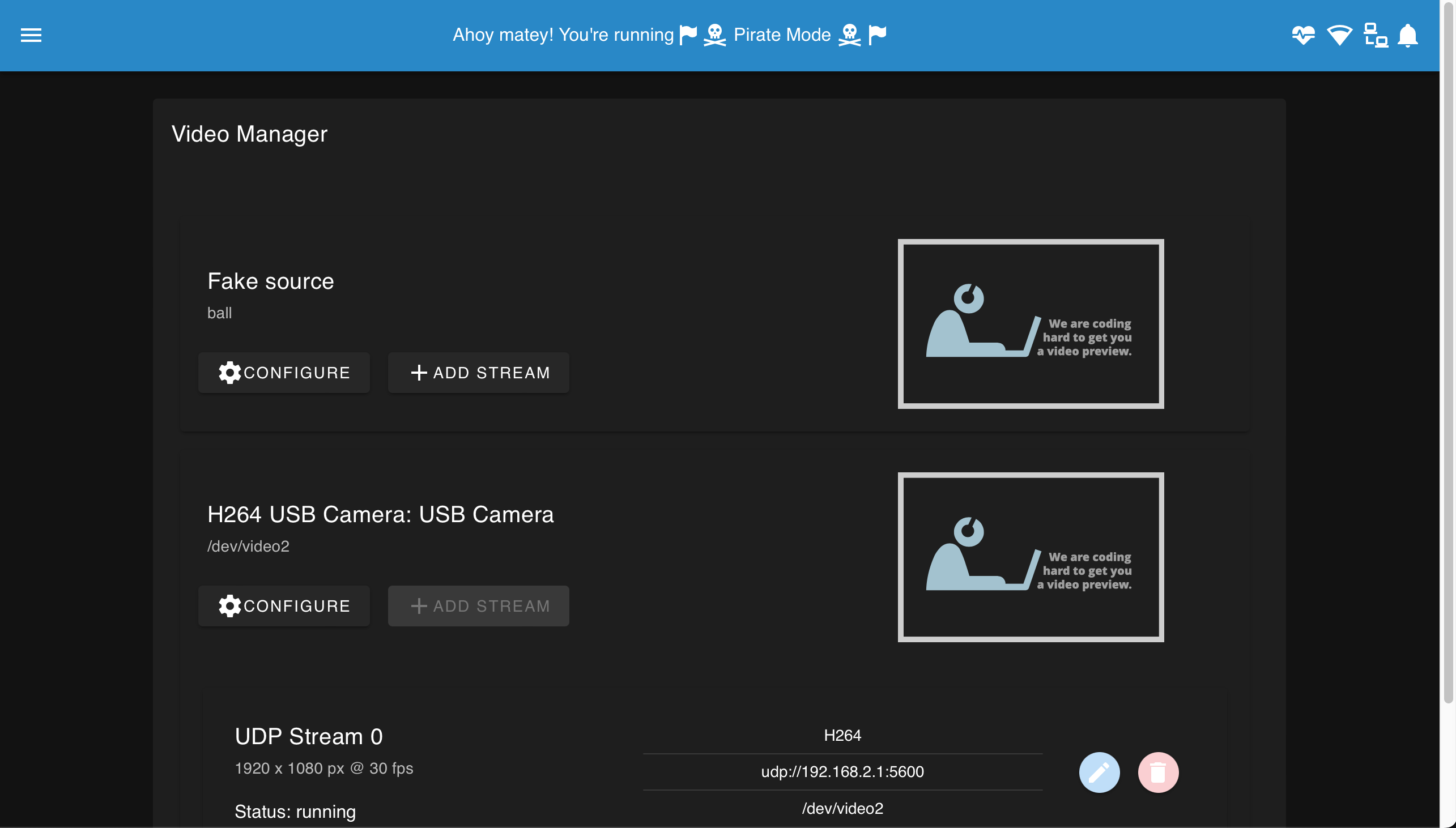
### Video

Based On: [*MAVLink Camera Manager*](https://github.com/bluerobotics/mavlink-camera-manager/) | Port:6020

* BlueOS automatically detects H264-encoded video streams
* The first time BlueOS starts up it will auto-configure any cameras that are connected at that time, with UDP streams counting up from port 5600
  + e.g. a second camera at first startup would be streamed to port 5601
  + Auto-configuration also occurs if the [settings are reset](https://docs.bluerobotics.com/ardusub-zola/software/onboard/BlueOS-1.0/advanced-usage/" \l "settings)
* It's also possible to manually reset only the camera settings by deleting the file /root/.config/mavlink-camera-manager/settings.json via [the file browser](https://docs.bluerobotics.com/ardusub-zola/software/onboard/BlueOS-1.0/advanced-usage/" \l "file-browser) or [the terminal](https://docs.bluerobotics.com/ardusub-zola/software/onboard/BlueOS-1.0/advanced-usage/" \l "terminal), or starting the camera manager inside the tmux session with the --reset flag
* After the initial startup, settings are saved and persistent across reboots
  + Further changes require manually re-configuring streams
  + New streams need to be manually added
    - The stream endpoint should be set to udp://<surface-IP>:<port>
    - e.g. udp://192.168.2.1:5601
* The streams are also presented via MAVLink, so QGroundControl (>=v4.1.7) can toggle between them without needing to know specific ports.



* Camera settings (brightness, exposure, etc) that are exposed via UVC can be configured with the "Configure" button
* Switching streams in QGroundControl while recording stops the current recording
  + If you are regularly switching streams it may be worth doing a screen recording either instead of or as well as recording the base video
* QGroundControl does not yet support displaying multiple streams simultaneously
  + Additional streams can be processed/viewed/recorded by [the options discussed here](https://discuss.bluerobotics.com/t/how-to-stream-another-cameras-video/9573/3" \l "receiving-the-stream-7)
* Raspberry Pi cameras are not yet supported in the pre-built releases/images
  + It is possible to do a custom installation on Raspberry Pi OS Buster or manually enable the legacy camera stack on Bullseye if necessary.



### (MAVLink) Endpoints

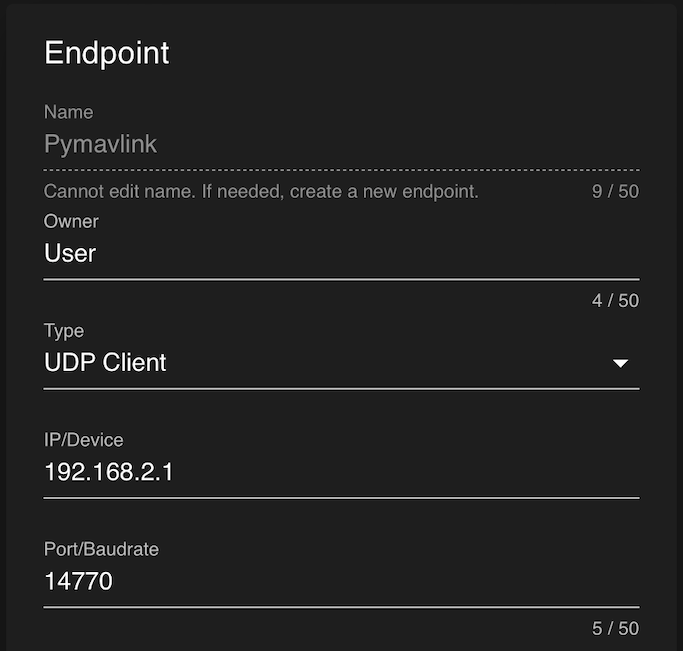
Based On: [*ArduPilot Manager*](https://github.com/bluerobotics/BlueOS-docker/tree/1.0.1/core/services/ardupilot_manager) | Port:8000

The endpoint manager allows managing the serial and UDP MAVLink endpoints and routing configurations.

* Endpoints intended for internal BlueOS operations are configured to the loopback IP 127.0.0.1
* Server endpoints for external use are configured to the localhost IP (e.g. 0.0.0.0; 192.168.2.2 may also work)
* Client endpoints for external use are configured to the external IP (e.g. 192.168.2.1)
* Client endpoints seem to operate more stably than server ones
* Unprotected endpoints can be removed or disabled:



* Modifying an endpoint is not possible - a new one must be created instead
  + e.g. some users may wish to set up a UDP endpoint for connecting to with Pymavlink from the surface:



**ROS Noetic Documentation**

## **ROS Noetic installation steps for Ubuntu 20.04 LTS Linux**

The steps we are performing in this installation guide can also be used on other Linux distros based on Ubuntu 20.04 Focal such as Linux Mint. However, at the time of writing this article, ROS doesn’t support Ubuntu Jammy, hence 22.04 users have to wait. Nevertheless, if it is necessary one can look for the [Source code installation](http://wiki.ros.org/noetic/Installation/Source) of this program.

**1. Open Terminal window.**

**2. Update and upgrade all softwares and then install curl.**

sudo apt update

sudo apt upgrade

sudo apt install curl -y

### **3. Add ROS Noetic package repository**

sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb\_release -sc) main" > /etc/apt/sources.list.d/ros-latest.list'

### **4. Add ROS Noetic Key on Ubuntu 20.04 and update.**

curl -s https://raw.githubusercontent.com/ros/rosdistro/master/ros.asc | sudo apt-key add -

sudo apt update

### **5. Installing ROS Noetic on Ubuntu 20.04**

sudo apt install ros-noetic-desktop-full

### **6. Setup Environment Variable for ROS**

echo "source /opt/ros/noetic/setup.bash" >> ~/.bashrc

source ~/.bashrc

we have to “source /opt/ros/noetic/setup.bash” this file ‘setup.bash’ everytime we make changes to our ROS package.

### **7. Rosdep commands**

sudo apt install python3-rosdep python3-rosinstall python3-rosinstall-generator python3-wstool build-essential

sudo rosdep update

sudo rosdep init

sudo rosdep fix-permissions

**For Help with Rosdep commands:**

rosedep --help

**ROSDEP are the dependancies used ina a ROS Package. These dependencies are needed to be installed before running a Pre-built ROS Package from github.**

# **Run Multiple Linux Terminal Sessions Simultaneously With Terminator**

sudo apt install terminator

You can also use the **Shift + Ctrl + O** keyboard shortcut to split the window horizontally and **Shift + Ctrl + E** to split it vertically. To navigate between different terminals, use the **Alt + arrow keys**.

**Terminator is Useful in Managing multiple running terminals while using ROS Noetic.**

**ROS Noetic Tutorials**

# **1. Creating a ROS Package:**

**For a package to be considered a catkin package it must meet a few requirements:**

* The package must contain a [catkin compliant package.xml](http://wiki.ros.org/catkin/package.xml) file. 
  + That package.xml file provides meta information about the package.
* The package must contain a [CMakeLists.txt which uses catkin](http://wiki.ros.org/catkin/CMakeLists.txt). 
  + If it is a [catkin metapackage](http://wiki.ros.org/catkin/package.xml" \l "Metapackages) it must have the relevant boilerplate CMakeLists.txt file.
* Each package must have its own folder 
  + This means no nested packages nor multiple packages sharing the same directory.

So, the Package will be contained in src folder of the Workspace.

**Create Workspace(use mkdir to create folder or you can create these folders manually):**

$ mkdir -p ~/catkin\_ws/src

(#copy the text after $ symbol. This symbol represents that this line is a terminal command.)

**Change directory to the workspace, create a package and build the workspace:**

$ cd ~/catkin\_ws

$ catkin\_create\_pkg beginner\_tutorials std\_msgs rospy roscpp

(#It creates a package named “beginner tutorials” with dependencies like roscpp, rospy and std\_msgs.)

$ catkin\_make

(#to build the workspace)

**To add the workspace to your ROS environment you need to source the generated setup file:**

$ . ~/catkin\_ws/devel/setup.bash

# **2. Understanding ROS Nodes**

This tutorial introduces ROS graph concepts and discusses the use of [roscore](http://wiki.ros.org/roscore), [rosnode](http://wiki.ros.org/rosnode), and [rosrun](http://wiki.ros.org/rosrun) commandline tools.

## **Quick Overview of Graph Concepts**

* [Nodes](http://wiki.ros.org/Nodes): A node is an executable that uses ROS to communicate with other nodes. A node really isn't much more than an executable file within a ROS package. ROS nodes use a ROS client library to communicate with other nodes. Nodes can publish or subscribe to a Topic. Nodes can also provide or use a Service.
* [Messages](http://wiki.ros.org/Messages): ROS data type used when subscribing or publishing to a topic.
* [Topics](http://wiki.ros.org/Topics): Nodes can publish messages to a topic as well as subscribe to a topic to receive messages.
* [Master](http://wiki.ros.org/Master): Name service for ROS (i.e. helps nodes find each other)
* [rosout](http://wiki.ros.org/rosout): ROS equivalent of stdout/stderr
* [roscore](http://wiki.ros.org/roscore): Master + rosout + parameter server (parameter server will be introduced later)

## **Client Libraries**

ROS client libraries allow nodes written in two programming languages to communicate:

* rospy = python client library
* roscpp = c++ client library

**roscore**

roscore is the first thing you should run when using ROS Think of it as an engine for ros to work on. roscore = ros+core : master (provides name service for ROS) + rosout (stdout/stderr) + parameter server (parameter server will be introduced later). Command:

$ roscore

If you get a permission error after running roscore you can use the command below:

$ sudo chown -R <your\_username> ~/.ros

## Using rosnode

Open up a new terminal, and let's use rosnode to see what running roscore did... Bear in mind to keep the previous terminal open either by opening a new tab or simply minimizing it.

rosnode displays information about the ROS nodes that are currently running. The rosnode list command lists these active nodes:

$ rosnode list

you will see a list of nodes that are currently active. The rosnode info command returns information about a specific node that is currently running.

$ rosnode info /rosout

This gave us some more information about /rosout node, such as the fact that it publishes /rosout\_agg.

## **3. Using rosrun**

rosrun allows you to use the package name to directly run a node within a package (without having to know the package path). Usage:

$ rosrun [package\_name] [node\_name] #This is a Template command it wont work in terminal.

For seeing the Execution Example of this command install tutorials package of ROS neotic:

$ sudo apt-get install ros-noetic-ros-tutorials

So now we can run the turtlesim\_node in the turtlesim package now:

$ rosrun turtlesim turtlesim\_node

You will see the turtlesim window. Now, in a new terminal window you can check the turtlesim node running using rosnode list.

One powerful feature of ROS is that you can reassign Names from the command-line.

Close the turtlesim window to stop the node (or go back to the rosrun turtlesim terminal and use ctrl-C). Now let's re-run it, but this time use a [Remapping Argument](http://wiki.ros.org/Remapping Arguments) to change the node's name:

$ rosrun turtlesim turtlesim\_node \_\_name:=my\_turtle

We see our new /my\_turtle node. Let's use another rosnode command, ping, to test that it's up:

$ rosnode ping my\_turtle

### **turtle keyboard teleoperation**

We'll also need something to drive the turtle around with. Please run in a new terminal:

$ rosrun turtlesim turtle\_teleop\_key

Now you can use the arrow keys of the keyboard to drive the turtle around. If you can not drive the turtle select the terminal window of the turtle\_teleop\_key to make sure that the keys that you type are recorded. Now that you can drive your turtle around, let's look at what's going on behind the scenes.

## **4. ROS Topics**

The turtlesim\_node and the turtle\_teleop\_key node are communicating with each other over a ROS Topic. turtle\_teleop\_key is publishing the key strokes on a topic, while turtlesim subscribes to the same topic to receive the key strokes. Let's use [rqt\_graph](http://wiki.ros.org/rqt_graph) which shows the nodes and topics currently running. To install rqt\_graph run:

$ sudo apt-get install ros-<distro>-rqt

$ sudo apt-get install ros-<distro>-rqt-common-plugins

# replacing <distro> with the name of your [ROS distribution](http://wiki.ros.org/Distributions) or ROS version name(e.g. indigo, jade, kinetic, lunar, noetic, etc.) like the below code:

$ sudo apt-get install ros-noetic-rqt

$ sudo apt-get install ros-noetic-rqt-common-plugins

$ sudo apt-get install ros-neotic-turtlesim

Now run:

$ rosrun rqt\_graph rqt\_graph



If you place your mouse cursor over /turtle1/command\_velocity it will highlight the ROS nodes (here blue and green) and topics (here red). As you can see, the turtlesim\_node and the turtle\_teleop\_key nodes are communicating on the topic named /turtle1/command\_velocity.

### **Introducing rostopic**

The rostopic tool allows you to get information about ROS topics.

$ rostopic -h

Output:

rostopic bw display bandwidth used by topic

rostopic echo print messages to screen

rostopic hz display publishing rate of topic

rostopic list print information about active topics

rostopic pub publish data to topic

rostopic type print topic type

Let's look at the command velocity data published by the turtle\_teleop\_key node. In a new Terminal run:

$ rostopic list

(#It will show a list of active topics. Lets look at The topic ending with /cmd\_vel. cmd\_vel stands for command velocity.)

$ rostopic echo /turtle1/cmd\_vel

You probably won't see anything happen because no data is being published on the topic. Let's make turtle\_teleop\_key publish data by pressing the arrow keys. Remember if the turtle isn't moving you need to select the turtle\_teleop\_key terminal again. You will see coordinates for linear and angular velocity. Lets move forward by running this next command in a new terminal:

$ rostopic list -h

Usage: rostopic list [/topic]

Options:

-h, --help show this help message and exit

-b or --bag list topics in .bag file

-v, --verbose list full details about each topic

-p list only publishers

-s list only subscribers

Use the verbose option.

$ rostopic list -v

(#It will show all Published Topics and subscribed topics with their publishers and subscribers.)

rostopic type returns the message type of any topic being published. Run:

$ rostopic type /turtle1/cmd\_vel

You Should get:

geometry\_msgs/Twist

## **5. ROS Messages**

Communication on topics happens by sending ROS messages between nodes. For the publisher (turtle\_teleop\_key) and subscriber (turtlesim\_node) to communicate, the publisher and subscriber must send and receive the same type of message. This means that a topic type is defined by the message type published on it. The type of the message sent on a topic can be determined using rostopic type. We can look at the details of the message using rosmsg:

$ rosmsg show geometry\_msgs/Twist

You Should get:

geometry\_msgs/Vector3 linear

float64 x

float64 y

float64 z

geometry\_msgs/Vector3 angular

float64 x

float64 y

float64 z

### **Using rostopic pub**

Now that we have learned about ROS messages, let's use rostopic with messages.

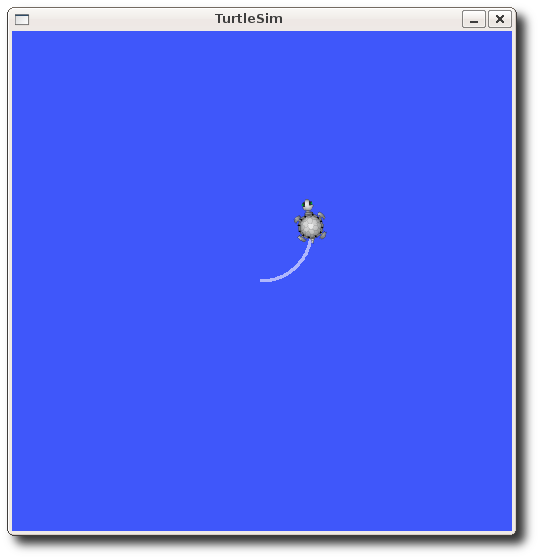
rostopic pub publishes data on to a topic currently advertised. Usage:

$ rostopic pub [topic] [msg\_type] [args]

Lets move forward with turtle to see how rostopic pub works:

$ rostopic pub -1 /turtle1/cmd\_vel geometry\_msgs/Twist -- '[2.0, 0.0, 0.0]' '[0.0, 0.0, 1.8]'

It would move the turtle like this in a quadrant curve:



-1 (dash-one) causes rostopic to only publish one message then exit.

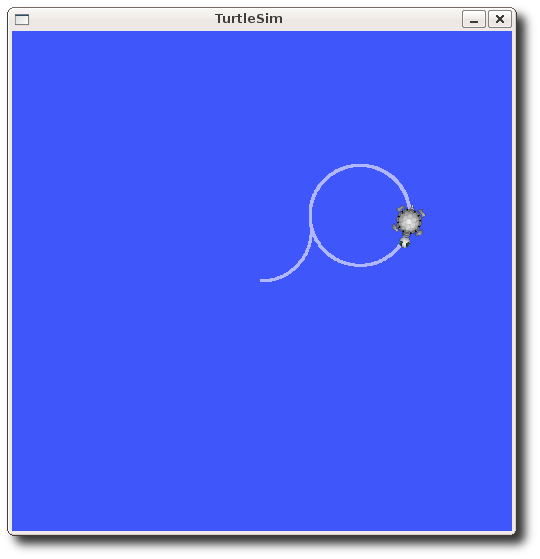
This -- (double-dash) in above command tells the option parser that none of the following arguments is an option. This is required in cases where your arguments have a leading dash -, like negative numbers.

As noted before, a geometry\_msgs/Twist msg has two vectors of three floating point elements each: linear and angular. In this case, '[2.0, 0.0, 0.0]' becomes the linear value with x=2.0, y=0.0, and z=0.0, and '[0.0, 0.0, 1.8]' is the angular value with x=0.0, y=0.0, and z=1.8. These arguments are actually in YAML syntax, which is described more in the [YAML command line documentation](http://wiki.ros.org/ROS/YAMLCommandLine).

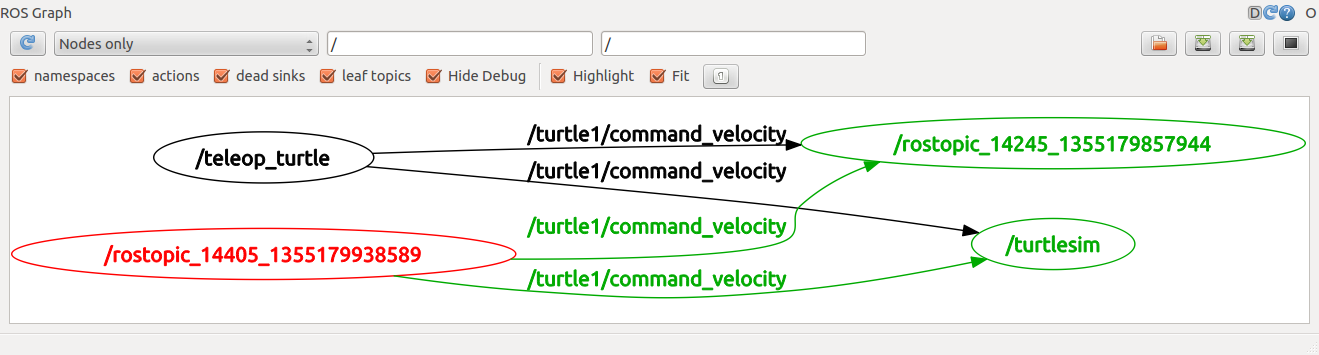
You may have noticed that the turtle has stopped moving; this is because the turtle requires a steady stream of commands at 1 Hz to keep moving. We can publish a steady stream of commands using rostopic pub -r command:

$ rostopic pub /turtle1/cmd\_vel geometry\_msgs/Twist -r 1 -- '[2.0, 0.0, 0.0]' '[0.0, 0.0, -1.8]'

This publishes the velocity commands at a rate of 1 Hz on the velocity topic.



We can also look at what is happening in rqt\_graph. Press the refresh button in the upper-left. The rostopic pub node (here in red) is communicating with the rostopic echo node (here in green):



As you can see the turtle is running in a continuous circle. In a new terminal, we can use rostopic echo to see the data published by our turtlesim:

$ rostopic echo /turtle1/pose

### Using rostopic hz

rostopic hz reports the rate at which data is published. Usage:

$ rostopic hz [topic]

Let's see how fast the turtlesim\_node is publishing /turtle1/pose:

$ rostopic hz /turtle1/pose

You will see:

subscribed to [/turtle1/pose]

average rate: 59.354

min: 0.005s max: 0.027s std dev: 0.00284s window: 58

average rate: 59.459

min: 0.005s max: 0.027s std dev: 0.00271s window: 118

average rate: 59.539

min: 0.004s max: 0.030s std dev: 0.00339s window: 177

average rate: 59.492

min: 0.004s max: 0.030s std dev: 0.00380s window: 237

average rate: 59.463

min: 0.004s max: 0.030s std dev: 0.00380s window: 290

Now we can tell that the turtlesim is publishing data about our turtle at the rate of 60 Hz. We can also use rostopic type in conjunction with rosmsg show to get in depth information about a topic:

$ rostopic type /turtle1/cmd\_vel | rosmsg show

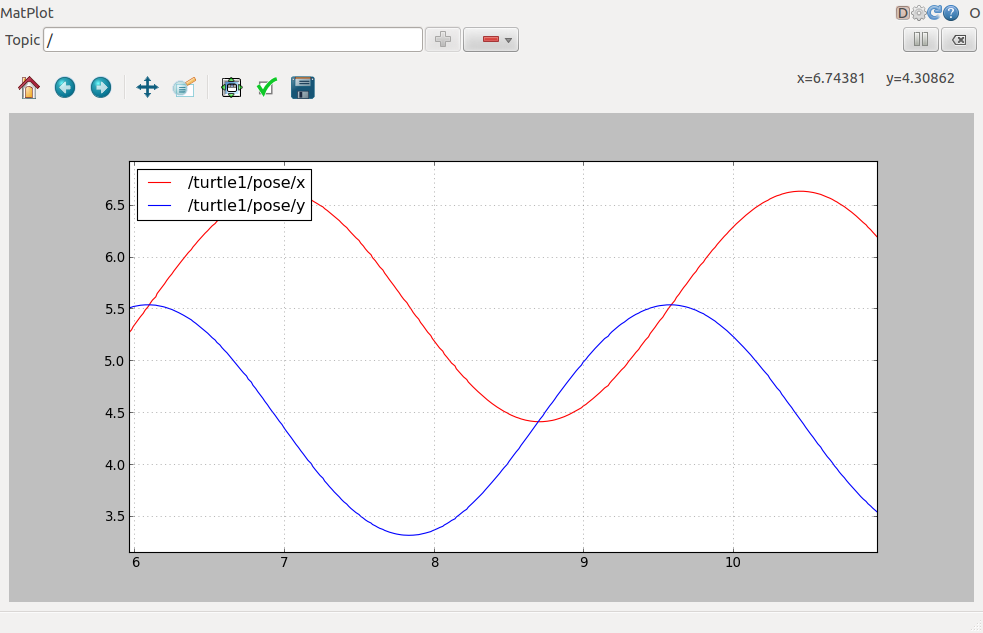
Now that we've examined the topics using rostopic let's use another tool to look at the data published by our turtlesim.

## **Using rqt\_plot**

rqt\_plot displays a scrolling time plot of the data published on topics. Here we'll use rqt\_plot to plot the data being published on the /turtle1/pose topic. First, start rqt\_plot by typing this command in a new terminal:

$ rosrun rqt\_plot rqt\_plot

In the new window that should pop up, a text box in the upper left corner gives you the ability to add any topic to the plot. Typing /turtle1/pose/x will highlight the plus button, previously disabled. Press it and repeat the same procedure with the topic /turtle1/pose/y. You will now see the turtle's x-y location plotted in the graph. Pressing the minus button shows a menu that allows you to hide the specified topic from the plot. Hiding both the topics you just added and adding /turtle1/pose/theta will result in the plot shown in the next figure.



## **6. ROS Services**

Services are another way that nodes can communicate with each other. Services allow nodes to send a request and receive a response.

## **Using rosservice**

rosservice can easily attach to ROS's client/service framework with services. rosservice has many commands that can be used on services, as shown below:

rosservice list print information about active services

rosservice call call the service with the provided args

rosservice type print service type

rosservice find find services by service type

rosservice uri print service ROSRPC uri

Now, Lets see all active services:

$ rosservice list

The list command shows us that the turtlesim node provides nine services: reset, clear, spawn, kill, turtle1/set\_pen, /turtle1/teleport\_absolute, /turtle1/teleport\_relative, turtlesim/get\_loggers, and turtlesim/set\_logger\_level. There are also two services related to the separate rosout node: /rosout/get\_loggers and /rosout/set\_logger\_level.

Let's look more closely at the clear service using rosservice type:

$ rosservice type /clear

You will get:

std\_srvs/Empty

This service is empty, this means when the service call is made it takes no arguments (i.e. it sends no data when making a request and receives no data when receiving a response). Let's call this service using rosservice call.

**Using rosservice call**

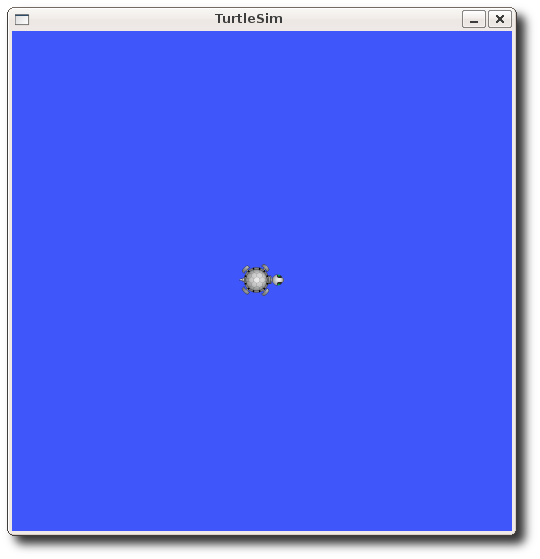
Usage:

rosservice call [service] [args]

Here we'll call with no arguments because the service is of type empty:

$ rosservice call /clear

This does what we expect, it clears the background of the turtlesim\_node.



Let's look at the case where the service has arguments by looking at the information for the service spawn:

$ rosservice type /spawn | rossrv show

you will get:

float32 x

float32 y

float32 theta

string name

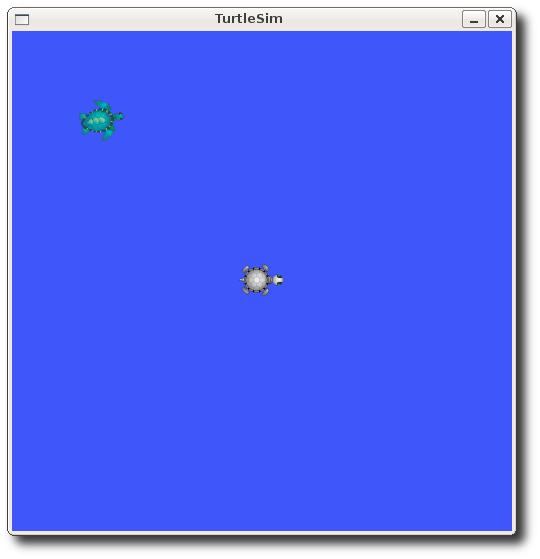
---

string name

This service lets us spawn a new turtle at a given location and orientation. The name field is optional, so let's not give our new turtle a name and let turtlesim create one for us.

$ rosservice call /spawn 2 2 0.2 ""

I hope you understand now that in the above code 2, 2 and 0.2 represents x, y and theta as you can see in the message type. The service call returns with the name of the newly created turtle. As the string name has not given a value it will go by default name as “turtle2”. Now our Turtlesim should look like this:



## **7. Using rosparam**

rosparam allows you to store and manipulate data on the ROS [Parameter Server](http://wiki.ros.org/Parameter Server). The Parameter Server can store integers, floats, boolean, dictionaries, and lists. rosparam uses the YAML markup language for syntax. In simple cases, YAML looks very natural: 1 is an integer, 1.0 is a float, one is a string, true is a boolean, [1, 2, 3] is a list of integers, and {a: b, c: d} is a dictionary. rosparam has many commands that can be used on parameters, as shown below:

rosparam set set parameter

rosparam get get parameter

rosparam load load parameters from file

rosparam dump dump parameters to file

rosparam delete delete parameter

rosparam list list parameter names

Let's look at what parameters are currently on the param server:

$ rosparam list

you will see:

/rosdistro

/roslaunch/uris/host\_nxt\_\_43407

/rosversion

/run\_id

/turtlesim/background\_b

/turtlesim/background\_g

/turtlesim/background\_r

Here we can see that the turtlesim node has three parameters on the param server for background color. Let's change one of the parameter values using rosparam set.

**rosparam set and rosparam get**

rosparam set [param\_name]

rosparam get [param\_name]

Now, we will change the parameter value and then we have to call the clear service for the parameter change to take effect. Here will change the red channel of the background color:

$ rosparam set /turtlesim/background\_r 150

$ rosservice call /clear

Now let's look at the values of other parameters on the param server. Let's get the value of the green background channel:

$ rosparam get /turtlesim/background\_g

We can also use rosparam get / to show us the contents of the entire Parameter Server.

$ rosparam get /

Output:

rosdistro: 'noetic

'

roslaunch:

uris:

host\_nxt\_\_43407: http://nxt:43407/

rosversion: '1.15.5

'

run\_id: 7ef687d8-9ab7-11ea-b692-fcaa1494dbf9

turtlesim:

background\_b: 255

background\_g: 86

background\_r: 69

You may wish to store this in a file so that you can reload it at another time. This is easy using rosparam. Usage:

$ rosparam dump [file\_name] [namespace]

$ rosparam load [file\_name] [namespace]

Here we write all the parameters to the file params.yaml

$ rosparam dump params.yaml

You can even load these yaml files into new namespaces, e.g. copy\_turtle:

$ rosparam load params.yaml copy\_turtle

$ rosparam get /copy\_turtle/turtlesim/background\_b

you will get 255 as an output.

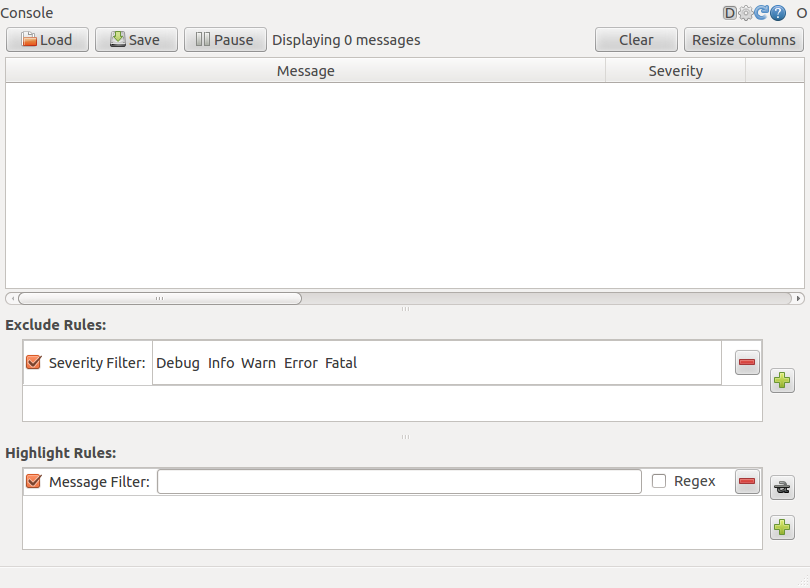
**8. Using rqt\_console and roslaunch**

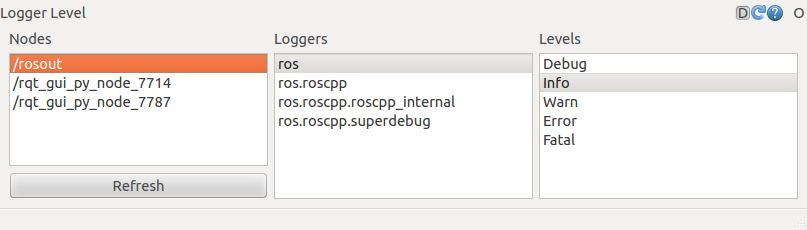
This tutorial introduces ROS using [rqt\_console](http://wiki.ros.org/rqt_console) and [rqt\_logger\_level](http://wiki.ros.org/rqt_logger_level) for debugging and [roslaunch](http://wiki.ros.org/roslaunch) for starting many nodes at once. rqt console attaches to ROS's logging framework to display output from nodes. rqt\_logger\_level allows us to change the verbosity level (DEBUG, WARN, INFO, and ERROR) of nodes as they run. Now let's look at the turtlesim output in rqt\_console and switch logger levels in rqt\_logger\_level as we use turtlesim. Before we start the turtlesim, in two new terminals start rqt\_console and rqt\_logger\_level:

$ rosrun rqt\_console rqt\_console

$ rosrun rqt\_logger\_level rqt\_logger\_level

You will see two windows popup:



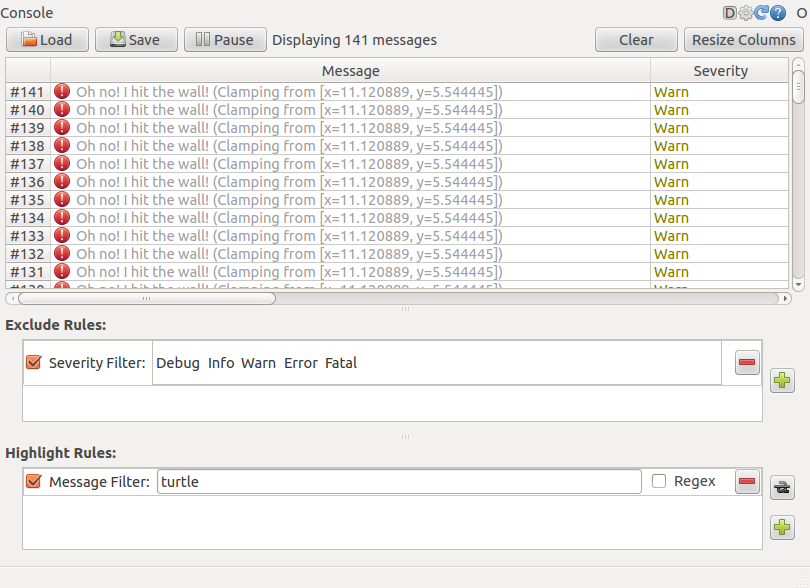


Now let's start turtlesim in a new terminal:

$ rosrun turtlesim turtlesim\_node

Since the default logger level is INFO you will see any info that the turtlesim publishes when it starts up. Now let's change the logger level to Warn by refreshing the nodes in the rqt\_logger\_level window and selecting Warn. Now let's run our turtle into the wall and see what is displayed in our rqt\_console.

$ rostopic pub /turtle1/cmd\_vel geometry\_msgs/Twist -r 1 -- '{linear: {x: 2.0, y: 0.0, z: 0.0}, angular: {x: 0.0,y: 0.0,z: 0.0}}'



Logging levels are prioritized in the following order:

Fatal

Error

Warn

Info

Debug

Fatal has the highest priority and Debug has the lowest. By setting the logger level, you will get all messages of that priority level or higher. For example, by setting the level to Warn, you will get all Warn, Error, and Fatal logging messages.

Let's Ctrl-C our turtlesim and let's use roslaunch to bring up multiple turtlesim nodes and a mimicking node to cause one turtlesim to mimic another.

**Using roslaunch**

roslaunch starts nodes as defined in a launch file. Usage:

$ roslaunch [package] [filename.launch]

Lets go to the beginner Tutorial package:

$ roscd beginner\_tutorials

If roscd cant find the package. Use The following commands:

$ cd ~/catkin\_ws

$ source devel/setup.bash

$ roscd beginner\_tutorials

Now, make a launch directory:

$ mkdir launch

$ cd launch

### The Launch File

Now let's create a launch file called turtlemimic.launch and paste the following:

[Toggle line numbers](http://wiki.ros.org/ROS/Tutorials/UsingRqtconsoleRoslaunch)

[1](http://wiki.ros.org/ROS/Tutorials/UsingRqtconsoleRoslaunch" \l "CA-91a3946a9c4cf7301bb55ec0c3f8a77f6c8f9777_1) <launch>

[2](http://wiki.ros.org/ROS/Tutorials/UsingRqtconsoleRoslaunch" \l "CA-91a3946a9c4cf7301bb55ec0c3f8a77f6c8f9777_2)

[3](http://wiki.ros.org/ROS/Tutorials/UsingRqtconsoleRoslaunch" \l "CA-91a3946a9c4cf7301bb55ec0c3f8a77f6c8f9777_3)  <group ns="turtlesim1">

[4](http://wiki.ros.org/ROS/Tutorials/UsingRqtconsoleRoslaunch" \l "CA-91a3946a9c4cf7301bb55ec0c3f8a77f6c8f9777_4)  <node pkg="turtlesim" name="sim" type="turtlesim\_node"/>

[5](http://wiki.ros.org/ROS/Tutorials/UsingRqtconsoleRoslaunch" \l "CA-91a3946a9c4cf7301bb55ec0c3f8a77f6c8f9777_5)  </group>

[6](http://wiki.ros.org/ROS/Tutorials/UsingRqtconsoleRoslaunch" \l "CA-91a3946a9c4cf7301bb55ec0c3f8a77f6c8f9777_6)

[7](http://wiki.ros.org/ROS/Tutorials/UsingRqtconsoleRoslaunch" \l "CA-91a3946a9c4cf7301bb55ec0c3f8a77f6c8f9777_7)  <group ns="turtlesim2">

[8](http://wiki.ros.org/ROS/Tutorials/UsingRqtconsoleRoslaunch" \l "CA-91a3946a9c4cf7301bb55ec0c3f8a77f6c8f9777_8)  <node pkg="turtlesim" name="sim" type="turtlesim\_node"/>

[9](http://wiki.ros.org/ROS/Tutorials/UsingRqtconsoleRoslaunch" \l "CA-91a3946a9c4cf7301bb55ec0c3f8a77f6c8f9777_9)  </group>

[10](http://wiki.ros.org/ROS/Tutorials/UsingRqtconsoleRoslaunch" \l "CA-91a3946a9c4cf7301bb55ec0c3f8a77f6c8f9777_10)

[11](http://wiki.ros.org/ROS/Tutorials/UsingRqtconsoleRoslaunch" \l "CA-91a3946a9c4cf7301bb55ec0c3f8a77f6c8f9777_11)  <node pkg="turtlesim" name="mimic" type="mimic">

[12](http://wiki.ros.org/ROS/Tutorials/UsingRqtconsoleRoslaunch" \l "CA-91a3946a9c4cf7301bb55ec0c3f8a77f6c8f9777_12)  <remap from="input" to="turtlesim1/turtle1"/>

[13](http://wiki.ros.org/ROS/Tutorials/UsingRqtconsoleRoslaunch" \l "CA-91a3946a9c4cf7301bb55ec0c3f8a77f6c8f9777_13)  <remap from="output" to="turtlesim2/turtle1"/>

[14](http://wiki.ros.org/ROS/Tutorials/UsingRqtconsoleRoslaunch" \l "CA-91a3946a9c4cf7301bb55ec0c3f8a77f6c8f9777_14)  </node>

[15](http://wiki.ros.org/ROS/Tutorials/UsingRqtconsoleRoslaunch" \l "CA-91a3946a9c4cf7301bb55ec0c3f8a77f6c8f9777_15)

[16](http://wiki.ros.org/ROS/Tutorials/UsingRqtconsoleRoslaunch" \l "CA-91a3946a9c4cf7301bb55ec0c3f8a77f6c8f9777_16) </launch>

### The Launch File Explained

Now, let's break the launch xml down.

[Toggle line numbers](http://wiki.ros.org/ROS/Tutorials/UsingRqtconsoleRoslaunch)

[1](http://wiki.ros.org/ROS/Tutorials/UsingRqtconsoleRoslaunch" \l "CA-21ef414cf4c910bb1286ff2aedfe349a32a099b9_1) <launch>

Here we start the launch file with the launch tag, so that the file is identified as a launch file.

[Toggle line numbers](http://wiki.ros.org/ROS/Tutorials/UsingRqtconsoleRoslaunch)

[3](http://wiki.ros.org/ROS/Tutorials/UsingRqtconsoleRoslaunch" \l "CA-0975bc12bed743bd6d6b7cf5af4a7bc4bf2fdd64_3)  <group ns="turtlesim1">

[4](http://wiki.ros.org/ROS/Tutorials/UsingRqtconsoleRoslaunch" \l "CA-0975bc12bed743bd6d6b7cf5af4a7bc4bf2fdd64_4)  <node pkg="turtlesim" name="sim" type="turtlesim\_node"/>

[5](http://wiki.ros.org/ROS/Tutorials/UsingRqtconsoleRoslaunch" \l "CA-0975bc12bed743bd6d6b7cf5af4a7bc4bf2fdd64_5)  </group>

[6](http://wiki.ros.org/ROS/Tutorials/UsingRqtconsoleRoslaunch" \l "CA-0975bc12bed743bd6d6b7cf5af4a7bc4bf2fdd64_6)

[7](http://wiki.ros.org/ROS/Tutorials/UsingRqtconsoleRoslaunch" \l "CA-0975bc12bed743bd6d6b7cf5af4a7bc4bf2fdd64_7)  <group ns="turtlesim2">

[8](http://wiki.ros.org/ROS/Tutorials/UsingRqtconsoleRoslaunch" \l "CA-0975bc12bed743bd6d6b7cf5af4a7bc4bf2fdd64_8)  <node pkg="turtlesim" name="sim" type="turtlesim\_node"/>

[9](http://wiki.ros.org/ROS/Tutorials/UsingRqtconsoleRoslaunch" \l "CA-0975bc12bed743bd6d6b7cf5af4a7bc4bf2fdd64_9)  </group>

Here we start two groups with a namespace tag of turtlesim1 and turtlesim2 with a turtlesim node with a name of sim. This allows us to start two simulators without having name conflicts.

[Toggle line numbers](http://wiki.ros.org/ROS/Tutorials/UsingRqtconsoleRoslaunch)

[11](http://wiki.ros.org/ROS/Tutorials/UsingRqtconsoleRoslaunch" \l "CA-78308b822a594630211cae0b2b508b405d07b108_11)  <node pkg="turtlesim" name="mimic" type="mimic">

[12](http://wiki.ros.org/ROS/Tutorials/UsingRqtconsoleRoslaunch" \l "CA-78308b822a594630211cae0b2b508b405d07b108_12)  <remap from="input" to="turtlesim1/turtle1"/>

[13](http://wiki.ros.org/ROS/Tutorials/UsingRqtconsoleRoslaunch" \l "CA-78308b822a594630211cae0b2b508b405d07b108_13)  <remap from="output" to="turtlesim2/turtle1"/>

[14](http://wiki.ros.org/ROS/Tutorials/UsingRqtconsoleRoslaunch" \l "CA-78308b822a594630211cae0b2b508b405d07b108_14)  </node>

Here we start the mimic node with the topics input and output renamed to turtlesim1 and turtlesim2. This renaming will cause turtlesim2 to mimic turtlesim1.

[Toggle line numbers](http://wiki.ros.org/ROS/Tutorials/UsingRqtconsoleRoslaunch)

[16](http://wiki.ros.org/ROS/Tutorials/UsingRqtconsoleRoslaunch" \l "CA-e33435967e0cc6e3273fdf1ce957aba2daaf5023_16) </launch>

This closes the xml tag for the launch file.

Now let's roslaunch the launch file:

$ roslaunch beginner\_tutorials turtlemimic.launch

Two turtlesims will start and in a new terminal send the rostopic command:

$ rostopic pub /turtlesim1/turtle1/cmd\_vel geometry\_msgs/Twist -r 1 -- '[2.0, 0.0, 0.0]' '[0.0, 0.0, -1.8]'

We can also use [rqt\_graph](http://wiki.ros.org/rqt_graph) to better understand what our launch file did. Run [rqt](http://wiki.ros.org/rqt)'s main window and select Plugins > Introspection > Node Graph Or simply use rqt\_graph.

# 9. Using rosed to edit files in ROS

**Using rosed**

rosed is part of the [rosbash](http://wiki.ros.org/rosbash) suite. It allows you to directly edit a file within a package by using the package name rather than having to type the entire path to the package. Usage:

$ rosed [package\_name] [filename]

Example:

$ rosed roscpp Logger.msg

This will open logger.msg file in vim if it is not opening maybe vim is not installed in your system. To exit or quit from vim enter q!, Changes would not be saved.

If the filename is not uniquely defined within the package, a menu will prompt you to choose which of the possible files you want to edit.

$ rosed roscpp <tab><tab>

It will show a list of possible files. The default editor for rosed is vim.

The more beginner-friendly editor nano is included with the default Ubuntu install. You can use it by editing your ~/.bashrc file to include:

$ export EDITOR='nano -w'

To set the default editor to emacs you can edit your ~/.bashrc file to include:

$ export EDITOR='emacs -nw'

NOTE: changes in .bashrc will only take effect for new terminals. Terminals that are already open will not see the new environmental variable.

Open a new terminal and see if EDITOR is defined:

$ echo $EDITOR

It will return the current Editor in Ubuntu.