ROS

Enjoy this cheat sheet at its fullest within Dash, the macOS documentation browser.

Filesystem Command-Line Tools

`apt-cache search ros-indigo`	Search for available packages on Ubuntu
`rospack/rosstack`	A tool inspecting packages/stacks. http://wiki.ros.org/rospack Usage: `rospack find [package]`
`roscd`	Changes directories to a package or stack. http://wiki.ros.org/rosbash Usage: `roscd [package[/subdir]]`
`rosls`	Lists package or stack information. http://wiki.ros.org/rosbash Usage: `rosls [package[/subdir]]`
`roscreate-pkg`	Creates a new ROS package. http://wiki.ros.org/roscreate Usage: `roscreate-pkg [package name]`
`roscreate-stack`	Creates a new ROS stack. http://wiki.ros.org/roscreate Usage: `roscreate-stack [path]`
`rosdep`	Installs ROS package system dependencies. http://wiki.ros.org/rosdep Usage: `rosdep install [package]`
`rosmake`	Builds a ROS package. http://wiki.ros.org/rosmake Usage: `rosmake [package]`
`roswtf`	Displays errors and warnings about a running ROS system or launch file.

	http://wiki.ros.org/roswtf Usage: `roswtf or roswtf [file]`

Common Command-Line Tools

`roscore`	A collection of nodes and programs that are pre-requisites of a ROS-based system. You must have a roscore running in order for ROS nodes to communicate. http://wiki.ros.org/roscore Usage: `roscore`
`rosmsg`	The rosmsg command-line tool displays information about ROS message types. http://wiki.ros.org/rosmsg Usage: `rosmsg [options]`
`rossrv`	The rossrv command-line tool displays information about ROS services. http://wiki.ros.org/rosmsg Usage: `rossrv [options]`
`rosrun`	The rosrun allows you to run an executable in an arbitrary package from anywhere without having to give its full path. http://wiki.ros.org/rosbash Usage: `rosrun package executable`
`rosnode`	Displays debugging information about ROS nodes, including publications, subscriptions and connections. http://wiki.ros.org/rosnode Usage: `rosnode [options]`
`roslaunch`	Starts ROS nodes locally and remotely via SSH, as well as setting parameters on the parameter server. http://wiki.ros.org/roslaunch Usage: `roslaunch [options]`
`rostopic`	A tool for displaying debug information about ROS topics, including publishers, subscribers, publishing rate, and messages. http://wiki.ros.org/rostopic

	Usage: `rostopic [options]`
`rosparam`	A tool for getting and setting ROS parameters on the parameter server using YAML-encoded files. http://wiki.ros.org/rosparam Usage: `rosparam [options]`
`rosservice`	A tool for listing and querying ROS services. http://wiki.ros.org/rosservice Usage: `rosservice [options]`

Logging Command-Line Tools

`rosbag`	This is a set of tools for recording from and playing back to ROS topics. It is intended to be high performance and avoids deserialization and reserialization of the messages. http://wiki.ros.org/rosbag Usage: `rosbag`

Graphical Tools

`rosgraph`	Displays a graph of the ROS nodes that are currently running, as well as the ROS topics that connect them. http://wiki.ros.org/rosgraph Usage: `rosgraph`
`rqt`	rqt is a Qt-based framework for GUI development for ROS. http://wiki.ros.org/rqt Usage: `rqt`
`rqt_bag`	rqt_bag provides a GUI plugin for displaying and replaying ROS bag files. http://wiki.ros.org/rqt_bag Usage: `rqt_bag`
`rqt_consol`	rqt_console provides a GUI plugin for displaying and filtering ROS messages. http://wiki.ros.org/rqt_consol Usage: `rqt_consol`

tf Command-Line Tools

`rosrun tf`	A tool that prints the information about a particular transformation between a source frame and a target frame. http://wiki.ros.org/tf Usage: `rosrun tf [options]]`

Workspaces

Create workspace

```
mkdir catkin_ws
cd catkin_ws
wstool init src
catkin_make
source devel/setup.bash
```

Add repo to workspace

```
roscd
cd ../src
wstool set repo_name --git http://github.com/org/repo_name.git --
version=indigo-devel
wstool up
```

Resolve dependencies in workspace

```
sudo rosdep init # only once
rosdep update
rosdep install --from-paths src --ignore-src --rosdistro=indigo -y
```

Packages

Create a package

```
catkin_create_pkg package_name [dependencies ...]
```

Package folders

```
include/package_name    # C++ header files
src                     # Source files, Python libraries in subdirectories
scripts                 # Python nodes and scripts
msg, srv, action        # Message, Service, and Action definitions
```

Release repo packages

```
catkin_generate_changelog
# review & commit changelogs"
catkin_prepare_release
bloom-release --track indigo --ros-distro indigo repo_name
```

CMakeLists.txt

Skeleton

```
cmake_minimum_required(VERSION 2.8.3)
project(package_name)
find_package(catkin REQUIRED)
catkin_package()
```

Package dependencies

```
find_package(catkin REQUIRED COMPONENTS roscpp)
catkin_package(
  INCLUDE_DIRS include
  LIBRARIES ${PROJECT_NAME}
  CATKIN_DEPENDS roscpp)
```

To use headers or libraries in a package, or to use a package's exported CMake macros, express a build-time dependency. Tell dependent packages what headers or libraries to pull in when your package is declared as a `catkin` component. Note that any packages listed as `CATKIN_DEPENDS` dependencies must also be declared as a `<run_depend>` in `package.xml`.

Messages, services

```
find_package(catkin REQUIRED COMPONENTS message_generation std_msgs)
add_message_files(FILES MyMessage.msg)
add_service_files(FILES MyService.msg)
generate_messages(DEPENDENCIES std_msgs)
catkin_package(CATKIN_DEPENDS message_runtime std_msgs)
```

These go after `find_package()`, but before `catkin_package()`.

Build libraries, executables

```
add_library(${PROJECT_NAME} src/main)
add_executable(${PROJECT_NAME}_node src/main)
target_link_libraries(${PROJECT_NAME}_node ${catkin_LIBRARIES})
```

These go after the `catkin_package()` call.

Installation

```
install(TARGETS ${PROJECT_NAME} DESTINATION
${CATKIN_PACKAGE_LIB_DESTINATION})
install(TARGETS ${PROJECT_NAME}_node DESTINATION
${CATKIN_PACKAGE_BIN_DESTINATION})
install(PROGRAMS scripts/myscript DESTINATION
${CATKIN_PACKAGE_BIN_DESTINATION})
install(DIRECTORY launch DESTINATION ${CATKIN_PACKAGE_SHARE_DESTINATION})
```

These go after the `catkin_package()` call.

Running System

Run ROS using plain

```
roscore
```

Running `roslaunch` will run its own `roscore` automatically

```
roslaunch my_package package_launchfile.launch
```

Nodes, topics, messages

```
rosnode list
rostopic list
rostopic echo cmd_vel
rostopic hz cmd_vel
rostopic info cmd_vel
rosmsg show geometry_msgs/Twist
```

Remote connection - master’s ROS environment

```
export ROS_IP or ROS_HOSTNAME set to this machine’s network address
export ROS_MASTER_URI set to URI containing that IP or hostname
```

Remote connection - your environment

```
export ROS_IP or ROS_HOSTNAME set to your machine’s network address
export ROS_MASTER_URI set to the URI from the master
```

To debug, check ping from each side to the other, run `roswtf` on each side

ROS Console

```
vi $HOME/.ros/config/rosconsole.config
  log4j.logger.ros.package_name=DEBUG
```

Adjust using `rqt_logger_level` and monitor via `rqt_console`. Use the `roslaunch --screen` flag to force all node output to the screen, as if each declared `<node>` had the `output="screen"` attribute.

Developer Commands

`catkin_make`	**Build all projects in workspace** Run from root folder. Example: `~/catkin_ws/`.
`catkin_make clean`	**Clean all projects in workspace** Run from root folder. Example: `~/catkin_ws/`.

You can modify and improve this cheat sheet here