# Managing Dependencies with rosdep

Table of Contents



**Goal:** Manage external dependencies using `rosdep`.

**Tutorial level:** Intermediate

**Time:** 5 minutes

This tutorial will explain how to manage external dependencies using `rosdep`.

> **Warning**
>
> Currently rosdep only works on Linux and macOS; Windows is not supported. There are long-term plans to add support for Windows to https://github.com/ros-infrastructure/rosdep.

# What is rosdep?

`rosdep` is a dependency management utility that can work with packages and external libraries. It is a command-line utility for identifying and installing dependencies to build or install a package. `rosdep` is *not* a package manager in its own right; it is a meta-package manager that uses its own knowledge of the system and the dependencies to find the appropriate package to install on a particular platform. The actual installation is done using the system package manager (e.g. `apt` on Debian/Ubuntu, `dnf` on Fedora/RHEL, etc).

It is most often invoked before building a workspace, where it is used to install the dependencies of the packages within that workspace.

It has the ability to work over a single package or over a directory of packages (e.g. workspace).

> **Note**
>
> While the name suggests it is for ROS, `rosdep` is semi-agnostic to ROS. You can utilize this powerful tool in non-ROS software projects by installing it as a standalone Python package. Successfully running `rosdep` relies on `rosdep keys` to be available, which can be downloaded from a public git repository with a few simple commands.

# A little about package.xml files

The `package.xml` is the file in your software where `rosdep` finds the set of dependencies. It is important that the list of dependencies in the `package.xml` is complete and correct, which allows all of the tooling to determine the packages dependencies. Missing or incorrect dependencies can lead to users not being able to use your package, to packages in a workspace being built out-of-order, and to packages not being able to be released.

The dependencies in the `package.xml` file are generally referred to as “rosdep keys”. These dependencies are manually populated in the `package.xml` file by the package’s creators and should be an exhaustive list of any non-builtin libraries and packages it requires.

These are represented in the following tags (see REP-149 for the full specification):

`<depend>`

These are dependencies that should be provided at both build time and run time for your package. For C++ packages, if in doubt, use this tag. Pure Python packages generally don’t have a build phase, so should never use this and should use `<exec_depend>` instead.

### `<build_depend>`

If you only use a particular dependency for building your package, and not at execution time, you can use the `<build_depend>` tag.

With this type of dependency, an installed binary of your package does not require that particular package to be installed.

However, that can create a problem if your package exports a header that includes a header from this dependency. In that case you also need a `<build_export_depend>`.

### `<build_export_depend>`

If you export a header that includes a header from a dependency, it will be needed by other packages that `<build_depend>` on yours. This mainly applies to headers and CMake configuration files. Library packages referenced by libraries you export should normally specify `<depend>`, because they are also needed at execution time.

### `<exec_depend>`

This tag declares dependencies for shared libraries, executables, Python modules, launch scripts and other files required when running your package.

### `<test_depend>`

This tag declares dependencies needed only by tests. Dependencies here should *not* be duplicated with keys specified by `<build_depend>`, `<exec_depend>`, or `<depend>`.

## How does rosdep work?

`rosdep` will check for `package.xml` files in its path or for a specific package and find the rosdep keys stored within. These keys are then cross-referenced against a central index to find the appropriate ROS package or software library in various package managers. Finally, once the packages are found, they are installed and ready to go!

`rosdep` works by retrieving the central index on to your local machine so that it doesn't have to access the network every time it runs (on Debian/Ubuntu the configuration for it is stored in `/etc/ros/rosdep/sources.list.d/20-default.list`).

The central index is known as `rosdistro`, which may be found online. We'll explore that more in the next section.

## How do I know what keys to put in my package.xml?

Great question, I'm glad you asked!

- If the package you want to depend in your package is ROS-based, AND has been released into the ROS ecosystem [1], e.g. `nav2_bt_navigator`, you may simply use the name of the package. You can find a list of all released ROS packages in https://github.com/ros/rosdistro at `<distro>/distribution.yaml` (e.g. `humble/distribution.yaml`) for your given ROS distribution.
- If you want to depend on a non-ROS package, something often called "system dependencies", you will need to find the keys for a particular library. In general, there are two files of interest:
  - rosdep/base.yaml contains the `apt` system dependencies
  - rosdep/python.yaml contains the Python dependencies

To find a key, search for your library in these files and find the name. This is the key to put in a `package.xml` file.

For example, imagine a package had a dependency on `doxygen` because it is a great piece of software that cares about quality documentation (hint hint). We would search `rosdep/base.yaml` for `doxygen` and come across:

```
doxygen:
  arch: [doxygen]
  debian: [doxygen]
  fedora: [doxygen]
  freebsd: [doxygen]
  gentoo: [app-doc/doxygen]
  macports: [doxygen]
  nixos: [doxygen]
  openembedded: [doxygen@meta-oe]
  opensuse: [doxygen]
  rhel: [doxygen]
  ubuntu: [doxygen]
```

That means our rosdep key is `doxygen`, which would resolve to those various names in different operating system's package managers for installation.

## What if my library isn't in rosdistro?

If your library isn't in `rosdistro`, you can experience the greatness that is open-source software development: you can add it yourself! Pull requests for rosdistro are typically merged well within a week.

Detailed instructions may be found here for how to contribute new rosdep keys. If for some reason these may not be contributed openly, it is possible to fork rosdistro and maintain a alternate index for use.

# How do I use the rosdep tool?

## rosdep installation

If you are using `rosdep` with ROS, it is conveniently packaged along with the ROS distribution. This is the recommended way to get `rosdep`. You can install it with:

```
apt-get install python3-rosdep
```

> **Note**
>
> On Debian and Ubuntu, there is another, similarly named package called `python3-rosdep2`. If that package is installed, make sure to remove it before installing `python3-rosdep`.

If you are using `rosdep` outside of ROS, the system package may not be available. In that case, you can install it directly from https://pypi.org:

```
pip install rosdep
```

## rosdep operation

Now that we have some understanding of `rosdep`, `package.xml`, and `rosdistro`, we're ready to use the utility itself! Firstly, if this is the first time using `rosdep`, it must be initialized via:

```
sudo rosdep init
rosdep update
```

This will initialize rosdep and `update` will update the locally cached rosdistro index. It is a good idea to `update` rosdep on occasion to get the latest index.

Finally, we can run `rosdep install` to install dependencies. Typically, this is run over a workspace with many packages in a single call to install all dependencies. A call for that would appear as the following, if in the root of the workspace with directory `src` containing source code.

```
rosdep install --from-paths src -y --ignore-src
```

Breaking that down:

- `--from-paths src` specifies the path to check for `package.xml` files to resolve keys for
- `-y` means to default yes to all prompts from the package manager to install without prompts
- `--ignore-src` means to ignore installing dependencies, even if a rosdep key exists, if the package itself is also in the workspace.

There are additional arguments and options available. Use `rosdep -h` to see them, or look at the more complete documentation for rosdep at http://docs.ros.org/en/independent/api/rosdep/html/ .

[1] "released into the ROS ecosystem" means the package is listed in one or more of the `<distro>/distribution.yaml` directories in the rosdistro database.