Please note that this blog post has old information that may no longer be correct. We invite you to read the content as a starting point but please search for more updated information in the ROS documentation <https://docs.ros.org/> .

A well configured linter can catch common errors before code is even run or compiled. ROS 2 makes it easy to add linters of your choice and make them part of your package's testing pipeline.

We'll step through the process, from start to finish, of adding a linter to ament so it can be used to automatically test your projects. We'll try to keep it generic, but where we need to lean on an example we'll be referring to the linter we recently added <https://ubuntu.com/blog/linting-ros-2-packages-with-mypy> for **mypy**, a static type analyzer for Python. You can view the finished source code for `ament_mypy <https://github.com/ament/ament_lint/tree/master/ament_mypy>` and `ament_cmake_mypy <https://github.com/ament/ament_lint/tree/master/ament_cmake_mypy>` .

Design

We'll need to make sure our linter integrates into `ament`'s testing pipeline. Namely, this means writing CMake scripts to integrate with `ament_cmake_test` and `ament_lint_auto`.

We need to be able to generate a JUnit XML <https://www.ibm.com/support/knowledgecenter/en/SSUFAU_1.0.0/com.ibm.rsar.analysis.codereview.cobol.doc/topics/cac_useresults_junit.html> report for the Jenkins build farm to parse, as well as handle automatically excluding directories with `AMENT_IGNORE` files, so we'll need to write a wrapper script for our linter as well.

Overall, we'll need to write the following packages:

- `ament_[linter]`
    - CLI wrapper for linter
        - Collect files, ignore those in `AMENT_IGNORE` directories
        - Configure and call linter
        - Generate XML report
- `ament_cmake_[linter]`
    - Set of CMake scripts
        - `ament_[linter].cmake`
            - Function to invoke linter wrapper
        - `ament_cmake_[linter]-extras.cmake`
            - Script to hook into `ament_lint_auto`
            - Registered at build as the CONFIG_EXTRA argument to `ament_package`
        - `ament_[linter].cmake`
            - Hook script for `ament_lint`

Getting Started – Python

We'll start with making the `ament_[linter]` package.

We'll be using Python to write this package, so we'll add a `setup.py` file, and fill out some required fields. It's easiest to just take one from an existing linter and customize it. What it ends up containing will be specific to the linter you're adding, but for **mypy** it looks like this:

```
from setuptools import find_packages
from setuptools import setup

setup(
    name='ament_mypy',
    version='0.7.3',
    packages=find_packages(exclude=['test']),
    package_data={'': [
        'configuration/ament_mypy.ini',
    ]},
    install_requires=['setuptools'],
    zip_safe=False,
    author='Ted Kern',
    author_email='<email>',
    maintainer='Ted Kern',
    maintainer_email='<email>',
    url='https://github.com/ament/ament_lint',
    download_url='https://github.com/ament/ament_lint/releases',
    keywords=['ROS'],
    classifiers=[
        'Intended Audience :: Developers',
        'License :: OSI Approved :: Apache Software License',
        'Programming Language :: Python',
        'Topic :: Software Development',
    ],
    description='Check Python static typing using mypy.',
    long_description="""\
The ability to check code for user specified static typing with mypy.""",
    license='Apache License, Version 2.0',
    tests_require=['pytest', 'pytest-mock'],
    entry_points={
        'console_scripts': [
            'ament_mypy = ament_mypy.main:main',
        ],
    },
)
```

We'll of course need a `package.xml` file. We'll need to make sure it has an `<exec_depend>` on the linter's package name in ROSDistro <https://github.com/ros/rosdistro> . If its not there, you'll need to go through the process of adding it <https://github.com/ros/rosdistro/blob/master/CONTRIBUTING.md> . This is required in order to actually install the linter itself as a dependency of our new ament linter package; without it any tests using it in CI would fail. Here's what it looks like for **mypy**:

```
<?xml version="1.0"?>
<?xml-model href="http://download.ros.org/schema/package_format3.xsd" schematypens="http://www.w3.org/2001/XMLSchema"?>
<package format="3">
  <name>ament_mypy</name>
  <version>0.7.3</version>
  <description>Support for mypy static type checking in ament.</description>
  <maintainer email="me@example.com">Ted Kern</maintainer>
  <license>Apache License 2.0</license>
  <author email="me@example.com">Ted Kern</author>

  <exec_depend>python3-mypy</exec_depend>

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

The Code

Create a python file called `ament_[linter]/main.py`, which will house all the logic for this linter. Below is the sample skeleton of a linter, again attempting to be generic where possible but nonetheless based on `ament_mypy`:

```
#!/usr/bin/env python3

import argparse
import os
import re
import sys
import textwrap
import time
from typing import List, Match, Optional, Tuple
from xml.sax.saxutils import escape
from xml.sax.saxutils import quoteattr

# Import your linter here
import mypy.api  # type: ignore

def main(argv: Optional[List[str]] = None) -> int:
    if not argv:
        argv = []

    parser.add_argument(
    'paths',
    nargs='*',
    default=[os.curdir],
    help='The files or directories to check. For directories files ending '
          'in '.py' will be considered.'
    )
    parser.add_argument(
        '--exclude',
        metavar='filename',
        nargs='*',
        dest='excludes',
        help='The filenames to exclude.'
    )
    parser.add_argument(
        '--xunit-file',
        help='Generate a xunit compliant XML file'
    )

    # Example of a config file specification option
    parser.add_argument(
        '--config',
        metavar='path',
        dest='config_file',
        default=os.path.join(os.path.dirname(__file__), 'configuration', 'ament_mypy.ini'),
        help='The config file'
    )

    # Example linter specific option
    parser.add_argument(
        '--cache-dir',
        metavar='cache',
        default=os.devnull,
        dest='cache_dir',
        help='The location mypy will place its cache in. Defaults to system '
             'null device'
    )

    args = parser.parse_args(argv)

    if args.xunit_file:
        start_time = time.time()

    if args.config_file and not os.path.exists(args.config_file):
        print("Could not find config file '{}'".format(args.config_file), file=sys.stderr)
        return 1

    filenames = _get_files(args.paths)
    if args.excludes:
        filenames = [f for f in filenames
                     if os.path.basename(f) not in args.excludes]
    if not filenames:
        print('No files found', file=sys.stderr)
        return 1

    normal_report, error_messages, exit_code = _generate_linter_report(
        filenames,
        args.config_file,
        args.cache_dir
    )

    if error_messages:
        print('mypy error encountered', file=sys.stderr)
        print(error_messages, file=sys.stderr)
        print('\nRegular report continues:')
        print(normal_report, file=sys.stderr)
        return exit_code

    errors_parsed = _get_errors(normal_report)

    print('\n{} files checked'.format(len(filenames)))
    if not normal_report:
        print('No errors found')
    else:
        print('{} errors'.format(len(errors_parsed)))

    print(normal_report)

    print('\nChecked files:')
    print(''.join(['\n* {}'.format(f) for f in filenames]))

    # generate xunit file
    if args.xunit_file:
        folder_name = os.path.basename(os.path.dirname(args.xunit_file))
        file_name = os.path.basename(args.xunit_file)
        suffix = '.xml'
        if file_name.endswith(suffix):
            file_name = file_name[:-len(suffix)]
            suffix = '.xunit'
            if file_name.endswith(suffix):
                file_name = file_name[:-len(suffix)]
        testname = '{}.{}'.format(folder_name, file_name)

        xml = _get_xunit_content(errors_parsed, testname, filenames, time.time() - start_time)
        path = os.path.dirname(os.path.abspath(args.xunit_file))
        if not os.path.exists(path):
            os.makedirs(path)
        with open(args.xunit_file, 'w') as f:
            f.write(xml)

    return exit_code


def _generate_linter_report(paths: List[str],
                          config_file: Optional[str] = None,
                          cache_dir: str = os.devnull) -> Tuple[str, str, int]:
    """Replace this section with code specific to your linter"""
    pass


def _get_xunit_content(errors: List[Match],
                       testname: str,
                       filenames: List[str],
                       elapsed: float) -> str:
    xml = textwrap.dedent("""\
        <?xml version="1.0" encoding="UTF-8"?>
        <testsuite
        name="{test_name:s}"
        tests="{test_count:d}"
        failures="{error_count:d}"
        time="{time:s}"
        >
    """).format(
                test_name=testname,
                test_count=max(len(errors), 1),
                error_count=len(errors),
                time='{:.3f}'.format(round(elapsed, 3))
    )

    if errors:
        # report each linter error/warning as a failing testcase
        for error in errors:
            pos = ''
            if error.group('lineno'):
                pos += ':' + str(error.group('lineno'))
                if error.group('colno'):
                    pos += ':' + str(error.group('colno'))
            xml += _dedent_to("""\
                <testcase
                    name={quoted_name}
                    classname="{test_name}"
                >
                    <failure message={quoted_message}/>
                </testcase>
                """, '  ').format(
                    quoted_name=quoteattr(
                        '{0[type]} ({0[filename]}'.format(error) + pos + ')'),
                    test_name=testname,
                    quoted_message=quoteattr('{0[msg]}'.format(error) + pos)
                )
    else:
        # if there are no mypy problems report a single successful test
        xml += _dedent_to("""\
            <testcase
              name="mypy"
              classname="{}"
              status="No problems found"/>
            """, '  ').format(testname)

    # output list of checked files
    xml += '  <system-out>Checked files:{escaped_files}\n  </system-out>\n'.format(
        escaped_files=escape(''.join(['\n* %s' % f for f in filenames]))
    )

    xml += '</testsuite>\n'
    return xml


def _get_files(paths: List[str]) -> List[str]:
    files = []
    for path in paths:
        if os.path.isdir(path):
            for dirpath, dirnames, filenames in os.walk(path):
                if 'AMENT_IGNORE' in filenames:
                    dirnames[:] = []
                    continue
                # ignore folder starting with . or _
                dirnames[:] = [d for d in dirnames if d[0] not in ['.', '_']]
                dirnames.sort()

                # select files by extension
                for filename in sorted(filenames):
                    if filename.endswith('.py'):
                        files.append(os.path.join(dirpath, filename))
        elif os.path.isfile(path):
            files.append(path)
    return [os.path.normpath(f) for f in files]


def _get_errors(report_string: str) -> List[Match]:
    return list(re.finditer(r'^(?P<filename>([a-zA-Z]:)?([^:])+):((?P<lineno>\d+):)?((?P<colno>\d+):)?\ (?P<type>error|warning|note):\ (?P<msg>.*)$', report_string, re.MULTILINE))  # noqa: E501


def _dedent_to(text: str, prefix: str) -> str:
    return textwrap.indent(textwrap.dedent(text), prefix)

if __name__ == 'main':
    sys.exit(main(sys.argv[1:]))
```

We'll break this down into chunks.

Main Logic

We write the file as an executable and use the argparse library to parse the invocation, so we begin the file with the shebang:

```
#!/usr/bin/env python3
```

and end it with the **main** logic:

```
if __name__ == 'main':
    sys.exit(main(sys.argv[1:]))
```

to forward failure codes out of the script.

The `main()` function will host the bulk of the program's logic. Define it, and make sure the `entry_points` argument in `setup.py` points to it.

```
def main(argv: Optional[List[str]] = None) -> int:
    if not argv:
        argv = []
```

Notice the use of **type hints <https://www.python.org/dev/peps/pep-0484/>** , mypy will perform static type checking where possible and where these hints are designated.

Parsing the Arguments

We add the arguments to argparse that ament expects:

```
parser.add_argument(
    'paths',
    nargs='*',
    default=[os.curdir],
    help='The files or directories to check. For directories files ending '
         'in '.py' will be considered.'
)
parser.add_argument(
    '--exclude',
    metavar='filename',
    nargs='*',
    dest='excludes',
    help='The filenames to exclude.'
)
parser.add_argument(
    '--xunit-file',
    help='Generate a xunit compliant XML file'
)
```

We also include any custom arguments, or args specific to the linter. For example, for **mypy** we also allow the user to pass in a custom config file to the linter, with a pre-configured default already set up:

```
# Example of a config file specification option
parser.add_argument(
    '--config',
    metavar='path',
    dest='config_file',
    default=os.path.join(os.path.dirname(__file__), 'configuration', 'ament_mypy.ini'),
    help='The config file'
)

# Example linter specific option
parser.add_argument(
    '--cache-dir',
    metavar='cache',
    default=os.devnull,
    dest='cache_dir',
    help='The location mypy will place its cache in. Defaults to system '
            'null device'
)
```

Note: remember to include any packaged non-code files (like default configs) using a manifest or `package_data=` in `setup.py`.

Finally, parse and validate the args:

```
args = parser.parse_args(argv)

if args.xunit_file:
    start_time = time.time()

if args.config_file and not os.path.exists(args.config_file):
    print("Could not find config file '{}'".format(args.config_file), file=sys.stderr)
    return 1

filenames = _get_files(args.paths)
if args.excludes:
    filenames = [f for f in filenames
                 if os.path.basename(f) not in args.excludes]
if not filenames:
    print('No files found', file=sys.stderr)
    return 1
```

Aside: _get_files

You'll notice the call to the helper function `_get_files`, shown below. We use a snippet from the other linters to build up an explicit list of files to lint, in order to apply our exclusions and the AMENT_IGNORE behavior.

```
def _get_files(paths: List[str]) -> List[str]:
    files = []
    for path in paths:
        if os.path.isdir(path):
            for dirpath, dirnames, filenames in os.walk(path):
                if 'AMENT_IGNORE' in filenames:
                    dirnames[:] = []
                    continue
                # ignore folder starting with . or _
                dirnames[:] = [d for d in dirnames if d[0] not in ['.', '_']]
                dirnames.sort()

                # select files by extension
                for filename in sorted(filenames):
                    if filename.endswith('.py'):
                        files.append(os.path.join(dirpath, filename))
        elif os.path.isfile(path):
            files.append(path)
    return [os.path.normpath(f) for f in files]
```

Note that in the near future this and `_get_xunit_content` will hopefully be de-duplicated into the `ament_lint` package.

This function, when given a list of paths, expands out all files recursively and returns those `.py` files that don't belong in directories containing an AMENT_IGNORE file.

We exclude those files that are in the exclude argument list, and we return a failure from main if no files are left afterwards.

```
filenames = _get_files(args.paths)

if args.excludes:
    filenames = [f for f in filenames
                 if os.path.basename(f) not in args.excludes]

if not filenames:
    print('No files found', file=sys.stderr)
    return 1
```

Otherwise we pass those files, as well as relevant configuration arguments, to the linter.

Invoking the Linter

We call the linter using whatever API it exposes:

```
normal_report, error_messages, exit_code = _generate_linter_report(
    filenames,
    args.config_file,
    args.cache_dir
)
```

abstracted here with the following method signature:

```
def _generate_linter_report(paths: List[str],
                            config_file: Optional[str] = None,
                            cache_dir: str = os.devnull) -> Tuple[str, str, int]:
```

Recording the Output

Any failures the linter outputs are printed to `stdout`, while any internal linter errors go to `stderr` and return the (non-zero) exit code:

```
if error_messages:
    print('linter error encountered', file=sys.stderr)
    print(error_messages, file=sys.stderr)
    print('\nRegular report continues:')
    print(normal_report, file=sys.stderr)
    return exit_code
```

We collect each warning/error/note message emitted individually:

```
errors_parsed = _get_errors(normal_report)
```

We then report the errors to the user with something like:

```
print('\n{} files checked'.format(len(filenames)))
if not normal_report:
    print('No errors found')
else:
    print('{} errors'.format(len(errors_parsed)))

print(normal_report)

print('\nChecked files:')
print(''.join(['\n* {}'.format(f) for f in filenames]))
```

Generating JUnit XML Output

Here we generate an xml report write the file to disk in the requested location.

```
if args.xunit_file:
        folder_name = os.path.basename(os.path.dirname(args.xunit_file))
        file_name = os.path.basename(args.xunit_file)
        suffix = '.xml'
        if file_name.endswith(suffix):
            file_name = file_name[:-len(suffix)]
            suffix = '.xunit'
            if file_name.endswith(suffix):
                file_name = file_name[:-len(suffix)]
        testname = '{}.{}'.format(folder_name, file_name)

        xml = _get_xunit_content(errors_parsed, testname, filenames, time.time() - start_time)
        path = os.path.dirname(os.path.abspath(args.xunit_file))
        if not os.path.exists(path):
            os.makedirs(path)
        with open(args.xunit_file, 'w') as f:
            f.write(xml)
```

An example of a valid output XML to **the schema <https://www.ibm.com/support/knowledgecenter/en/SSUFAU_1.0.0/com.ibm.rsar.analysis.codereview.cobol.doc/topics/cac_useresults_junit.html>** is shown below

```
<?xml version="1.0" encoding="UTF-8"?>
<testsuite
name="tst"
tests="4"
failures="4"
time="0.010"
>
  <testcase
      name="error (/tmp/pytest-of-ubuntu/pytest-164/use_me7/lc.py:0:0)"
      classname="tst"
  >
      <failure message="error message:0:0"/>
  </testcase>
  <testcase
      name="error (/tmp/pytest-of-ubuntu/pytest-164/use_me7/l.py:0)"
      classname="tst"
  >
      <failure message="error message:0"/>
  </testcase>
  <testcase
      name="error (/tmp/pytest-of-ubuntu/pytest-164/use_me7/no_pos.py)"
      classname="tst"
  >
      <failure message="error message"/>
  </testcase>
  <testcase
      name="warning (/tmp/pytest-of-ubuntu/pytest-164/use_me7/warn.py)"
      classname="tst"
  >
      <failure message="warning message"/>
  </testcase>
  <system-out>Checked files:
* /tmp/pytest-of-ubuntu/pytest-164/use_me7/lc.py
* /tmp/pytest-of-ubuntu/pytest-164/use_me7/l.py
* /tmp/pytest-of-ubuntu/pytest-164/use_me7/no_pos.py
* /tmp/pytest-of-ubuntu/pytest-164/use_me7/warn.py
  </system-out>
</testsuite>
```

Aside: _get_xunit_content

We write a helper function, `_get_xunit_content`, to format the XML output to **the schema <https://www.ibm.com/support/knowledgecenter/en/SSUFAU_1.0.0/com.ibm.rsar.analysis.codereview.cobol.doc/topics/cac_useresults_junit.html>** . This one is a bit specific to **mypy**, but hopefully it gives you a good idea of what's needed:

```
def _get_xunit_content(errors: List[Match],
                       testname: str,
                       filenames: List[str],
                       elapsed: float) -> str:
    xml = textwrap.dedent("""\
        <?xml version="1.0" encoding="UTF-8"?>
        <testsuite
        name="{test_name:s}"
        tests="{test_count:d}"
        failures="{error_count:d}"
        time="{time:s}"
        >
    """).format(
                test_name=testname,
                test_count=max(len(errors), 1),
                error_count=len(errors),
                time='{:.3f}'.format(round(elapsed, 3))
    )

    if errors:
        # report each mypy error/warning as a failing testcase
        for error in errors:
            pos = ''
            if error.group('lineno'):
                pos += ':' + str(error.group('lineno'))
                if error.group('colno'):
                    pos += ':' + str(error.group('colno'))
            xml += _dedent_to("""\
                <testcase
                    name={quoted_name}
                    classname="{test_name}"
                >
                    <failure message={quoted_message}/>
                </testcase>
                """, '  ').format(
                    quoted_name=quoteattr(
                        '{0[type]} ({0[filename]}'.format(error) + pos + ')'),
                    test_name=testname,
                    quoted_message=quoteattr('{0[msg]}'.format(error) + pos)
                )
    else:
        # if there are no mypy problems report a single successful test
        xml += _dedent_to("""\
            <testcase
              name="mypy"
              classname="{}"
              status="No problems found"/>
            """, '  ').format(testname)

    # output list of checked files
    xml += '  <system-out>Checked files:{escaped_files}\n  </system-out>\n'.format(
        escaped_files=escape(''.join(['\n* %s' % f for f in filenames]))
    )

    xml += '</testsuite>\n'
    return xml
```

Return from main

Finally, we return the exit code.

```
return exit_code
```

The CMake Plugin

Now that our linting tool is ready, we need to write an interface for it to attach to ament.

Getting Started

We create a new ros 2 package named `ament_cmake_[linter]` in the `ament_lint` folder, and fill out `package.xml`. As an example, the one for **mypy** looks like this:

```
<?xml version="1.0"?>
<?xml-model href="http://download.ros.org/schema/package_format3.xsd" schematypens="http://www.w3.org/2001/XMLSchema"?>
<package format="3">
  <name>ament_cmake_mypy</name>
  <version>0.7.3</version>
  <description>
    The CMake API for ament_mypy to perform static type analysis on python code
    with mypy.
  </description>
  <maintainer email="<email>">Ted Kern</maintainer>
  <license>Apache License 2.0</license>
  <author email="<email>">Ted Kern</author>

  <buildtool_depend>ament_cmake_core</buildtool_depend>
  <buildtool_depend>ament_cmake_test</buildtool_depend>

  <buildtool_export_depend>ament_cmake_test</buildtool_export_depend>
  <buildtool_export_depend>ament_mypy</buildtool_export_depend>

  <test_depend>ament_cmake_copyright</test_depend>
  <test_depend>ament_cmake_lint_cmake</test_depend>

  <export>
    <build_type>ament_cmake</build_type>
  </export>
</package>
```

CMake Configuration

We write the installation and testing instructions in `CMakeLists.txt`, as well as pass our extras file to `ament_package`. This is the one for **mypy**, yours should look pretty similar:

```
cmake_minimum_required(VERSION 3.5)

project(ament_cmake_mypy NONE)

find_package(ament_cmake_core REQUIRED)
find_package(ament_cmake_test REQUIRED)

ament_package(
  CONFIG_EXTRAS "ament_cmake_mypy-extras.cmake"
)

install(
  DIRECTORY cmake
  DESTINATION share/${PROJECT_NAME}
)

if(BUILD_TESTING)
  find_package(ament_cmake_copyright REQUIRED)
  ament_copyright()

  find_package(ament_cmake_lint_cmake REQUIRED)
  ament_lint_cmake()
endif()
```

Then we register our extension with ament in `ament_cmake_[linter]-extras.cmake`. Again, this one is for **mypy**, but you should be able to easily repurpose it.

```
find_package(ament_cmake_test QUIET REQUIRED)

include("${ament_cmake_mypy_DIR}/ament_mypy.cmake")

ament_register_extension("ament_lint_auto" "ament_cmake_mypy"
  "ament_cmake_mypy_lint_hook.cmake")
```

We then create a CMake function in `cmake/ament_[linter].cmake` to invoke our test when needed. This will be specific to your linter and the wrapper you wrote above, but here’s how it looks for **mypy**:

```
#
# Add a test to statically check Python types using mypy.
#
# :param CONFIG_FILE: the name of the config file to use, if any
# :type CONFIG_FILE: string
# :param TESTNAME: the name of the test, default: "mypy"
# :type TESTNAME: string
# :param ARGN: the files or directories to check
# :type ARGN: list of strings
#
# @public
#
function(ament_mypy)
  cmake_parse_arguments(ARG "" "CONFIG_FILE;TESTNAME" "" ${ARGN})
  if(NOT ARG_TESTNAME)
    set(ARG_TESTNAME "mypy")
  endif()

  find_program(ament_mypy_BIN NAMES "ament_mypy")
  if(NOT ament_mypy_BIN)
    message(FATAL_ERROR "ament_mypy() could not find program 'ament_mypy'")
  endif()

  set(result_file "${AMENT_TEST_RESULTS_DIR}/${PROJECT_NAME}/${ARG_TESTNAME}.xunit.xml")
  set(cmd "${ament_mypy_BIN}" "--xunit-file" "${result_file}")
  if(ARG_CONFIG_FILE)
    list(APPEND cmd "--config-file" "${ARG_CONFIG_FILE}")
  endif()
  list(APPEND cmd ${ARG_UNPARSED_ARGUMENTS})

  file(MAKE_DIRECTORY "${CMAKE_BINARY_DIR}/ament_mypy")
  ament_add_test(
    "${ARG_TESTNAME}"
    COMMAND ${cmd}
    OUTPUT_FILE "${CMAKE_BINARY_DIR}/ament_mypy/${ARG_TESTNAME}.txt"
    RESULT_FILE "${result_file}"
    WORKING_DIRECTORY "${CMAKE_CURRENT_SOURCE_DIR}"
  )
  set_tests_properties(
    "${ARG_TESTNAME}"
    PROPERTIES
    LABELS "mypy;linter"
  )
endfunction()
```

This function checks for the existence of your linting CLI, prepares the argument list to pass in, creates an output directory for the report, and labels the test type.

Finally, in `ament_cmake_[linter]_lint_hook.cmake`, we write the hook into the function we just defined. This one is for **mypy** but yours should look almost identical:

```
file(GLOB_RECURSE _python_files FOLLOW_SYMLINKS "*.py")
if(_python_files)
  message(STATUS "Added test 'mypy' to statically type check Python code.")
  ament_mypy()
endif()
```

Final Steps

With both packages ready, we build our new packages using colcon:

```
~/ros2/src $ colcon build --packages-select ament_mypy ament_cmake_mypy --event-handlers console_direct+ --symlink-install
```

If all goes well, we can now use this linter just like any other to test our Python packages!

It's highly recommended you write a test suite to go along with your code. `ament_mypy` lints itself with flake8 and mypy, and has an extensive **pytest <https://docs.pytest.org/en/latest/contents.html>** based suite of functions to validate its behavior. You can see this suite **here <https://github.com/ament/ament_lint/blob/master/ament_mypy/test/test_ament_mypy.py>** .

Check out our **other article <https://ubuntu.com/blog/linting-ros-2-packages-with-mypy>** on how to use the **mypy** linter if you'd like to learn more about how to invoke linters from your testing suite for other packages.

Internet of Things

From home control to drones, robots and industrial systems, Ubuntu Core and Snaps provide robust security, app stores and reliable updates for all your IoT devices.

Get the latest Ubuntu news and updates in your inbox.

Work email: ________

☐ *I agree to receive information about Canonical's products and services.

By submitting this form, I confirm that I have read and agree to **Canonical's Privacy Policy <https://www.ubuntu.com/legal/dataprivacy>** .

Sign up

Are you building a robot on top of Ubuntu and looking for a partner? Talk to us!

Contact Us

Related posts

Optimise your ROS snap – Part 2

Welcome to Part 2 of the "optimise your ROS snap" blog series. Make sure to check Part 1 before reading this blog post. This second part is going to present...

Optimise your ROS snap – Part 1

Do you want to optimise the performance of your ROS snap? We reduced the size of the installed Gazebo snap by 95%! This is how you can do it for your snap....

ROS orchestration with snaps

Application orchestration is the process of integrating applications together to automate and synchronise processes. In robotics, this is essential,...