
ARCHIVED

null-ls is now archived and will no longer receive updates. Please see this issue for details.

null-ls.nvim

Use Neovim as a language server to inject LSP diagnostics, code actions, and more via Lua.

Motivation

Neovim's LSP ecosystem is growing, and plugins like telescope.nvim and trouble.nvim make it a joy to work with LSP features like code actions and diagnostics.

Unlike the VS Code and coc.nvim ecosystems, Neovim doesn't provide a way for non-LSP sources to hook into its LSP client. null-ls is an attempt to bridge that gap and simplify the process of creating, sharing, and setting up LSP sources using pure Lua.

null-ls is also an attempt to reduce the boilerplate required to set up general-purpose language servers and improve performance by removing the need for external processes.

Status

null-ls is in **beta status**. Please see below for steps to follow if something doesn't work the way you expect (or doesn't work at all).

null-ls is developed on and tested against the latest stable version of Neovim. Support for versions built from [HEAD](#) is provided on a best-effort basis, and users are encouraged to contribute fixes to any issues exclusive to these versions.

Features

null-ls sources are able to hook into the following LSP features:

- Code actions
- Diagnostics (file- and project-level)
- Formatting (including range formatting)
- Hover

-
- Completion

null-ls includes built-in sources for each of these features to provide out-of-the-box functionality. See BUILTINS for a list of available built-in sources and BUILTIN_CONFIG for instructions on how to set up and configure these sources.

null-ls also provides helpers to streamline the process of spawning and transforming the output of command-line processes into an LSP-friendly format. If you want to create your own source, either for personal use or for a plugin, see HELPERS for details.

Setup

Install null-ls using your favorite package manager. The plugin depends on plenary.nvim, which you are (probably) already using.

To get started, you must set up null-ls and register at least one source. See BUILTINS for a list of available built-in sources and CONFIG for information about setting up and configuring null-ls.

```
1 local null_ls = require("null-ls")
2
3 null_ls.setup({
4   sources = {
5     null_ls.builtins.formatting.stylua,
6     null_ls.builtins.diagnostics.eslint,
7     null_ls.builtins.completion.spell,
8   },
9 })
```

Documentation

The definitive source for information about null-ls is its documentation, which contains information about how null-ls works, how to set it up, and how to create sources.

Contributing

Contributions to add new features and built-ins for any language are always welcome. See CONTRIBUTING for guidelines.

Examples

Parsing buffer content

The following example demonstrates a diagnostic source that will parse the current buffer's content and show instances of the word `really` as LSP warnings.

```
1 local null_ls = require("null-ls")
2
3 local no_really = {
4     method = null_ls.methods.DIAGNOSTICS,
5     filetypes = { "markdown", "text" },
6     generator = {
7         fn = function(params)
8             local diagnostics = {}
9             -- sources have access to a params object
10            -- containing info about the current file and editor state
11            for i, line in ipairs(params.content) do
12                local col, end_col = line:find("really")
13                if col and end_col then
14                    -- null-ls fills in undefined positions
15                    -- and converts source diagnostics into the
16                    -- required format
17                    table.insert(diagnostics, {
18                        row = i,
19                        col = col,
20                        end_col = end_col + 1,
21                        source = "no-really",
22                        message = "Don't use 'really!'",
23                        severity = vim.diagnostic.severity.WARN,
24                    })
25                end
26            end
27            return diagnostics
28        end,
29    },
30 }
31 null_ls.register(no_really)
```

Parsing CLI program output

`null-ls` includes helpers to simplify the process of spawning and capturing the output of CLI programs. This example shows how to pass the content of the current buffer to `markdownlint` via `stdin` and convert its output (which it sends to `stderr`) into LSP diagnostics:

```
1 local null_ls = require("null-ls")
```

```

2  local helpers = require("null-ls.helpers")
3
4  local markdownlint = {
5      method = null_ls.methods.DIAGNOSTICS,
6      filetypes = { "markdown" },
7      -- null_ls.generator creates an async source
8      -- that spawns the command with the given arguments and options
9      generator = null_ls.generator({
10         command = "markdownlint",
11         args = { "--stdin" },
12         to_stdin = true,
13         from_stderr = true,
14         -- choose an output format (raw, json, or line)
15         format = "line",
16         check_exit_code = function(code, stderr)
17             local success = code <= 1
18
19             if not success then
20                 -- can be noisy for things that run often (e.g.
21                 -- diagnostics), but can
22                 -- be useful for things that run on demand (e.g.
23                 -- formatting)
24                 print(stderr)
25             end
26
27             return success
28         end,
29         -- use helpers to parse the output from string matchers,
30         -- or parse it manually with a function
31         on_output = helpers.diagnostics.from_patterns({
32             {
33                 pattern = [[:(%d+):( %d+) [%w-/+]+ (.*)]],
34                 groups = { "row", "col", "message" },
35             },
36             {
37                 pattern = [[:(%d+) [%w-/+]+ (.*)]],
38                 groups = { "row", "message" },
39             },
40         }),
41     },
42     null_ls.register(markdownlint)

```

FAQ

Something isn't working! What do I do?

NOTE: If you run into issues when using null-ls, please follow the steps below and **do not** open an issue on the Neovim repository. null-ls is not an actual LSP server, so we need to determine whether issues are specific to this plugin before sending anything upstream.

1. Make sure your configuration is in line with the latest version of this document.
2. Enable debug mode and check the output of your source(s). If the CLI program is not properly configured or is otherwise not running as expected, that's an issue with the program, not null-ls.
3. Check the documentation for available configuration options that might solve your issue.
4. If you're having trouble configuring null-ls or want to know how to achieve a specific result, open a discussion.
5. If you believe the issue is with null-ls itself or you want to request a new feature, open an issue and provide the information requested in the issue template.

My `:checkhealth` output is wrong! What do I do?

Checking whether a given command is executable is tricky, and null-ls' health check doesn't handle all cases. null-ls' internal command resolution is independent of its health check output, which is for informational purposes.

If you're not sure whether a given command is running as expected, enable debug mode and check your log.

How do I format files?

Use `vim.lsp.buf.format()`. See `:help vim.lsp.buf.format()` for usage instructions.

How do I format files on save?

See this wiki page.

How do I stop Neovim from asking me which server I want to use for formatting?

See this wiki page.

How do I view project-level diagnostics?

For a built-in solution, use `:lua vim.diagnostic.setqflist()`. You can also use a plugin like `trouble.nvim`.

How do I enable debug mode and get debug output?

1. Set `debug` flag to `true` in your config:

```
1 require("null-ls").setup({  
2   debug = true,  
3 })
```

2. Use `:NullLsLog` to open your debug log in the current Neovim instance or `:NullLsInfo` to get the path to your debug log.

As with LSP logging, debug mode will slow down Neovim. Make sure to disable the option after you've collected the information you're looking for.

Does it work with (other plugin)?

In most cases, yes. `null-ls` tries to act like an actual LSP server as much as possible, so it should work seamlessly with most LSP-related plugins. If you run into problems, please try to determine which plugin is causing them and open an issue.

This wiki page mentions plugins that require specific configuration options / tweaks to work with `null-ls`.

How does it work?

Thanks to hard work by @folke, the plugin wraps the mechanism Neovim uses to spawn language servers to start a client entirely in-memory. The client attaches to buffers that match defined sources and receives and responds to requests, document changes, and other events from Neovim.

Will it affect my performance?

More testing is necessary, but since `null-ls` uses pure Lua and runs entirely in memory without any external processes, in most cases it should run faster than similar solutions. If you notice that performance is worse with `null-ls` than with an alternative, please open an issue!

I am seeing a formatting timeout error message

This issue occurs when a formatter takes longer than the default timeout value. This is an automatic mechanism and controlled by Neovim. You might want to increase the timeout in your call:

```
1 vim.lsp.buf.format({ timeout_ms = 2000 })
```

Tests

The test suite includes unit and integration tests and depends on `plenary.nvim`. Run `make test` in the root of the project to run the suite or `FILE=filename_spec.lua make test-file` to test an individual file.

To avoid a dependency on any plugin managers, the test suite will set up its plugin runtime under the `./tests` directory to always have a plenary version available.

If you run into plenary-related issues while running the tests, make sure you have an up-to-date version of the plugin by clearing that cache with: `make clean`.

All tests expect to run on the latest release version of Neovim and are not guaranteed to work on versions built from `HEAD`.

Alternatives

- `efm-langserver` and `diagnostic-languageserver`: general-purpose language servers that can provide formatting and diagnostics from CLI output.
- `nvim-lint`: a Lua plugin that focuses on providing diagnostics from CLI output.
- `formatter.nvim`: a Lua plugin that (surprise) focuses on formatting.
- `hover.nvim`: Hover plugin framework for Neovim.